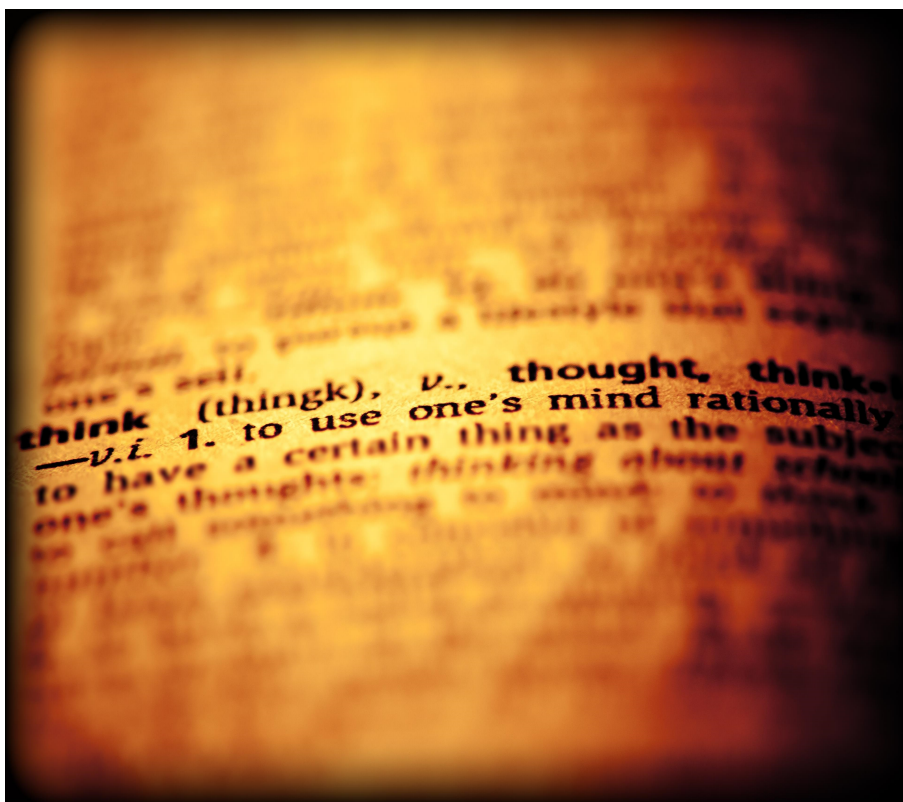
Istituto Statale d'Istruzione Superiore
"Arturo Malignani" - Udine

Istituto Tecnico Industriale "A. Malignani"
Sezione di Telecomunicazione ed Informatica
Dipartimento di Elettronica

Linguaggio C

Versione 0v64
Dicembre 2014
prof. Santino Bandiziol

Indice

Convenzioni adottate nel testo

Il testo normale è scritto utilizzando il presente carattere tipografico e stile di scrittura. Si è cercato di evitare inutili inglesismi ed un eccesso di acronimi, ma trattandosi di appunti tecnici è inevitabile il ricorso all'inglese tecnico e a frequenti abbreviazioni.

Nella prima eventualità, il termine in inglese incontrato per la prima volta viene scritto in corsivo, come nel caso della parola *font* (carattere tipografico). Nell'ipotesi in cui sia ritenuto utile, fra parentesi viene indicata una possibile traduzione che tenga conto del contesto. Le successive volte in cui la stessa parola viene riutilizzata, non verrà più evidenziata in corsivo, dando per acquisito il termine.

Quando lo si ritiene utile la presente convenzione viene usata a "rovescio", ponendo tra parentesi la traduzione inglese di un termine italiano, come nel caso in cui si voglia parlare del carattere tipografico (*font*) e fornirne la traduzione.

L'uso degli acronimi segue regole simili. Viene indicato l'acronimo in grassetto e tra parentesi il suo significato, come nel caso dell'acronimo **PC** (Personal Computer). A volte la parentesi è omessa, come nel caso in cui la spiegazione del significato venga fornita in forma discorsiva e non didascalica. Anche in questo caso, successivi usi dello stesso termine non prevedono nè il grassetto nè l'indicazione del significato posto fra parentesi.

Il corsivo viene utilizzato anche per termini e frasi in italiano che si intendono *enfatizzare*, come nel presente caso. Si è comunque cercato di non abusare di tale convenzione.

L'icona di pericolo è utilizzata per richiamare l'attenzione del lettore su un passaggio particolarmente importante. E' bene, quindi, leggere con attenzione quanto evidenziato dalla presenza del triangolo di pericolo.

Introduzione

Real Programmers aren't afraid to use GOTOs.
Real Programmers can write five page long DO loops without getting confused.
Real Programmers enjoy Arithmetic IF statements because they make the code more interesting.
Real Programmers write self-modifying code, especially if it saves them 20 nanoseconds in the middle of a tight loop.
Real Programmers don't need comments: the code is obvious.

Ed Post - 1982

I presenti appunti contengono alcune riflessioni sulla programmazione in linguaggio C, nonché alcuni esercizi. Essi si prefiggono lo scopo di aiutare l'allievo nel non facile compito di apprendere un linguaggio di programmazione tutt'altro che semplice e accrescere la propria autonomia nella risoluzione di problemi di natura logica.

Una prima scelta che si è dovuto operare ha riguardato lo standard di riferimento ed il linguaggio vero e proprio. Scelta non scontata dovendo optare fra ANSI C, C89, C99, C11, C++, C#, ecc. In conclusione si è deciso nel più ovvio dei modi, prendendo come riferimento l'ultimo documento ufficiale dei linguaggi C imperativi: lo standard internazionale IEC/ISO 9899:TC3 denominato *Programming languages - C*. La versione definitiva è disponibile presso l'*International Organisation for Standardization* (ISO) oppure presso l'*International Electrotechnical Commission* (IEC). La versione *Draft* è invece disponibile liberamente al seguente indirizzo:

http://www.open-std.org/JTC1/SC22/WG14/

Un'ultima riflessione. Gli appunti sono volutamente presentati in una forma grafica piuttosto piatta. Vi è la tendenza, oggidì, a "bordare" in grigio le definizioni, a evidenziare le parti importanti di un discorso e a porre in risalto lemmi e postulati, inibendo, parzialmente, in tal modo la capacità critica dell'allievo. E' come se l'autore dicesse al proprio lettore: "Non ti preoccupare: ti indico io le cose importanti. Poni particolare attenzione soltanto alle parti evidenziate e trascura pure quelle anonime".

Ciò spinge lo studente a soffermarsi (studiare a memoria?) solo su alcune parti, dimenticando dimostrazioni, riflessioni, approfondimenti, ecc., compromettendo in tal modo la solida costruzione del proprio sapere.

Si vuole, invece, far capire allo studente che studiare significa anche fare propria la conoscenza divulgata da altri e che ciò implica atteggiamento critico e impegno.

Le presenti pagine sono state scritte e redatte con cura. Ciò non significa che siano prive di errori o imprecisioni. Quanti volessero segnalare eventuali errori, possono farlo al seguente indirizzo:

bandiziol@katamail.com

Grazie e buon lavoro.

Udine, 27/12/2014

prof. Santino Bandiziol

Parte I

I PREREQUISITI

Capitolo 1

L'algoritmo

Prima di occuparci del linguaggio di programmazione trattato nelle presenti pagine è importante introdurre il concetto di **algoritmo**.

L'etimologia del termine è stata a lungo ambigua, ma sembra ormai certo che derivi dal matematico persiano *Abū 'Abd Allāh Muhammed ibn Mūsā al-Khwārizmi*, vissuto nel IX secolo nella regione del lago *Khwaārizm*, l'attuale lago Aral, e autore del trattato *Kitab al-jabr wa'l-muqābala* da cui deriva il termine "algebra".

Intuitivamente l'algoritmo è la descrizione di un insieme di azioni che, se eseguite, permettono di risolvere un dato problema. In tale ottica, l'esempio classico che si usa citare è la ricetta di cucina, che permette di ottenere, ad esempio, una torta, se si seguono determinate azioni partendo da determinati ingredienti.

Se, però, si vuole dare una definizione esaustiva del termine algoritmo si deve assumere un punto di vista un po' più rigoroso.

1.1 Le caratteristiche dell'algoritmo

Donald E. Knuth è da considerarsi uno dei padri dell'informatica moderna. Ha scritto una monumentale e preziosissima opera dal titolo eloquente: "The Art of Computer Programming" che è tuttora in fase di revisione e completamento. In essa l'autore identifica 5 caratteristiche fondamentali che ciascun algoritmo deve possedere:

- deve essere **finito** (*finitness*);
- deve essere **definito** (*definitness*);
- deve avere **dati d'ingresso enumerabili** (*input*);
- deve produrre **dati di uscita** (*output*);
- deve essere **effettivo** (*effectiveness*).

La mancanza di una sola delle suddette caratteristiche fa sì che l'algoritmo cessi di essere tale.

1.1.1 *Finitness*

L'algoritmo deve essere **finito**. Ciò significa che deve essere costituito da un numero finito di passi, ovvero di azioni, che si ripetono. Tale sequenza deve avere, prima o poi, un termine.

Ciò non significa, si badi bene, che in informatica non si possano ipotizzare cicli "infiniti", ma semplicemente che detti cicli non possono essere catalogati come algoritmi. Si pensi, ad esempio, ad un orologio elettronico. Sicuramente il cuore del programma che lo realizza è un ciclo che attende "all'infinito" lo scadere di un secondo, per aggiornare, a quel punto, l'ora corrente. La parte di programma relativa all'attesa che il secondo scada può considerarsi (dal punto di vista della *finitness*) un algoritmo, ma non il ciclo che contiene tale parte, dato che non prevede un numero finito di passi.

Una procedura che possiede le ultime 4 caratteristiche elencate precedentemente ma che non ha un numero di passi finito, è definita da Donald E. Knuth come un "metodo di calcolo" (cfr. KNUTH [8]).

1.1.2 *Definitness*

Ogni passo dell'algoritmo deve essere accuratamente **definito** senza ambiguità alcuna. A tal fine il linguaggio naturale non si presta sempre alla perfezione, proprio perché, per sua natura, il linguaggio usato quotidianamente per comunicare è ambiguo.

Si pensi ad esempio alle figure retoriche. Quello che nel linguaggio parlato è chiarissimo e da tutti interpretato correttamente (metonìmia: "facile come bere un bicchiere d'acqua") se interpretato alla lettera può apparire paradossale (non si può bere un bicchiere d'acqua: si beve dell'acqua contenuta in un bicchiere di vetro).

Naturalmente si può prestare attenzione nella formulazione degli enunciati che descrivono l'algoritmo, in modo da evitare le insidie del linguaggio naturale, ma, solitamente, si preferisce ricorrere ad alcuni strumenti di frequente uso nella definizione degli algoritmi: i diagrammi di flusso e gli pseudo-linguaggi. Esiste, infine, un terzo strumento capace di descrivere l'algoritmo in maniera perfetta senza la minima ambiguità: il linguaggio di programmazione.

La presente caratteristica è sicuramente quella che mette più frequentemente in imbarazzo lo studente. Se l'algoritmo presenta delle criticità o dei passi di una certa complessità, spesso lo studente è tentato di descriverli in maniera ambigua. Ciò va assolutamente evitato.

1.1.3 *Input*

L'algoritmo deve possedere zero o più **dati di ingresso** enumerabili, che possono essere forniti prima e/o durante la sua esecuzione.

Normalmente questa caratteristica appare piuttosto incomprensibile allo studente. Che senso ha dire che un algoritmo deve avere "zero o più" dati di ingresso, aggiungendo anche che se tale caratteristica viene meno l'algoritmo non è più tale?

La risposta sta nell'aggettivo "enumerabili". Non è importante quanti siano i dati d'ingresso (possono essere anche mancanti) ma devono essere enumerabili. La loro quantità, quindi, deve essere nota e non ambigua.

1.1.4 *Output*

L'algoritmo deve produrre uno o più **dati di uscita**. Anche in questo caso c'è spazio per aprire una discussione. I "dati di uscita" non sono necessariamente di natura numerica, ma possono avere le più svariate forme: suoni, immagini, colori, ecc.

Scherzosamente (ma non troppo) si potrebbe aggiungere che anche un algoritmo "che non fa niente" produce un dato d'uscita: perde tempo. Far scorrere una ben precisa quantità di tempo è spesso assolutamente fondamentale, ad esempio, nella realizzazione di un orologio.

1.1.5 *Effectiveness*

L'algoritmo deve essere **effettivo**. Ciò significa che i singoli passi devono essere tutti eseguibili.

Dire che il singolo passo deve essere eseguibile non significa solamente che "deve essere teoricamente possibile eseguirlo", ma anche che deve essere possibile eseguirlo in pratica. Descrivere un algoritmo che contenga dei passi che non sono eseguibili in pratica, significa invalidare l'intero algoritmo.

1.2 Definizione di algoritmo

Ora che le 5 caratteristiche che un algoritmo deve possedere sono state illustrate, si può tentare di definirlo, ovvero di darne una descrizione sintetica ma esaustiva.

Va specificato, innanzitutto, che l'algoritmo è considerato un concetto primitivo, e quindi non è formalmente definibile mediante altri concetti primitivi.

All'inizio del capitolo si è detto che l'algoritmo è un "insieme di regole". Le regole, però, servono a mettere ordine nelle azioni, quindi è più corretto parlare di "processo". Il processo in questione deve possedere le 5 caratteristiche spiegate nella sezione, quindi potrebbe essere fornita la seguente definizione:

L'algoritmo è un insieme finito di passi eseguibili e non ambigui che, a partire da dati di ingresso enumerabili, definiscono un processo che produce uno o più dati d'uscita.

1.3 Il buon algoritmo

Vale la pena aggiungere qualche riflessione in merito alle 5 caratteristiche che l'algoritmo deve possedere. La loro presenza è da considerarsi condizione necessaria non sufficiente per identificare un buon algoritmo. Perchè ciò che importa non è descrivere un algoritmo, ma descriverne uno efficiente.

Ideare un algoritmo che identifica il numero di passeggeri di un'automobile, ne valuta la disposizione sui sedili e l'effettivo uso delle cinture di sicurezza, calcola velocità, accelerazione e quantità di moto del mezzo con un errore massimo inferiore all'1% sui valori reali, per poi far intervenire gli *airbag* con due secondi di ritardo su un eventuale impatto, significa aver buttato al vento tempo e risorse.

Un buon algoritmo non deve solo essere efficiente, ma deve essere anche economico, ovvero deve coinvolgere il minor numero possibile di risorse, in modo tale che costo e prestazioni del sistema siano armoniosamente equilibrati fra loro. Ciò significa evitare sovrastrutture inutili o appensantimenti del codice. Tutto ciò appare evidente se si pensa soprattutto al *hardware* del sistema, ma deve essere oggetto di riflessione anche pensando al solo *software*.

Infine, un'ultima riflessione. Ogni buon algoritmo va **commentato** e contestualizzato. Vanno definite chiaramente le condizioni al contorno e gli obiettivi e, soprattutto, vanno illustrati con molta chiarezza i singoli passi e la struttura complessiva dell'algoritmo. Vanno evidenziati i punti critici o di non immediata comprensione e vanno illustrate le scelte prese e, soprattutto, "perché" sono state prese. Uno *slogan* efficace potrebbe essere il seguente:

Il linguaggio formale descrive il "come",
il commento spiega il "perché".

Infine, Donald E. Knuth fornisce un prezioso consiglio allo studente che si accosta a questa interessante materia, sostenendo che "gli algoritmi vanno eseguiti, non studiati". Con ciò intende dire che per capire un algoritmo bisogna applicarlo e non limitarsi semplicemente a studiarlo.

1.4 La congettura di Collatz

Si è detto nella sezione 1.1.2 che il linguaggio naturale non si presta alla perfezione per illustrare un algoritmo. Ciò nonostante, i prossimi esercizi che verranno proposti saranno presentati in tal modo, un po' per non anticipare gli strumenti formali che verranno presentati nelle prossime sezioni e un po' per sottolineare la necessaria precisione di linguaggio.

A tal proposito, lo studente dovrà sforzarsi di interpretare il corretto significato delle proposizioni che illustrano i singoli passi dell'algoritmo, evitando di falsarne il significato.

Il seguente esercizio *dovrebbe* essere un algoritmo. Si tratta della *Congettura di Collatz* che prende il nome dal matematico tedesco Lothar Collatz.

Esercizio

Sia dato il seguente algoritmo:

1. Sia dato un numero intero positivo n
2. Se n è dispari, si assegni ad n il risultato del calcolo $3n + 1$
3. Se n è pari si assegni ad n il risultato del calcolo $n/2$
4. Se n = 1 l'algoritmo termina, altrimenti si torna al punto 2

Secondo Collatz, il suddetto algoritmo giunge *sempre* a termine. Ciò, però, non è stato ancora dimostrato. Si applichi l'algoritmo, con carta e penna, per valori di n pari a 17, 100, 128, 27 (attenzione a quest'ultimo numero).

Soluzione

Come propone l'esercizio, si assegna ad n inizialmente il valore 17, per cui vengono eseguiti i seguenti calcoli:

#	Passo	Pari/Dispari	Esecuzione	Commento
01	2.	Dispari	$n = 3 \cdot 17 + 1 = 52$	Calcolo di $n = 3n + 1$
02	4.	/	$n \neq 1$	n è diverso da 1
03	3.	Pari	$n = 52/2 = 26$	Calcolo di $n = n/2$
04	4.	/	$n \neq 1$	n è diverso da 1
05	3.	Pari	$n = 26/2 = 13$	Calcolo di $n = n/2$
06	4.	/	$n \neq 1$	n è diverso da 1
07	2.	Dispari	$n = 3 \cdot 13 + 1 = 40$	Calcolo di $n = 3n + 1$
08	4.	/	$n \neq 1$	n è diverso da 1
09	3.	Pari	$n = 40/2 = 20$	Calcolo di $n = n/2$
10	4.	/	$n \neq 1$	n è diverso da 1
11	3.	Pari	$n = 20/2 = 10$	Calcolo di $n = n/2$
12	4.	/	$n \neq 1$	n è diverso da 1
13	3.	Pari	$n = 10/2 = 5$	Calcolo di $n = n/2$
14	4.	/	$n \neq 1$	n è diverso da 1
15	2.	Dispari	$n = 3 \cdot 5 + 1 = 16$	Calcolo di $n = 3n + 1$
16	4.	/	$n \neq 1$	n è diverso da 1
17	3.	Pari	$n = 16/2 = 8$	Calcolo di $n = n/2$
18	4.	/	$n \neq 1$	n è diverso da 1
19	3.	Pari	$n = 8/2 = 4$	Calcolo di $n = n/2$
20	4.	/	$n \neq 1$	n è diverso da 1
21	3.	Pari	$n = 4/2 = 2$	Calcolo di $n = n/2$
22	4.	/	$n \neq 1$	n è diverso da 1
23	3.	Pari	$n = 2/2 = 1$	Calcolo di $n = n/2$
24	4.	/	$n = 1$	n è uguale a 1: fine

Tabella 1.1: Congettura di Collatz

1.5 L'algoritmo di Euclide

Un algoritmo sicuramente più utile e più famoso è l'algoritmo di Euclide per il calcolo del MCD fra due numeri interi. Esso compare nel libro IX degli *Elementi* di Euclide (cfr. ACERBI [?]) e permette di calcolare velocemente il massimo comun divisore fra due numeri interi.

Esercizio

Sia dato, quindi, il seguente algoritmo, noto come algoritmo di Euclide per il calcolo del MCD:

1. Siano dati due numeri interi positivi n ed m con $n > m > 0$
2. Si calcoli il resto r della divisione di n per m
3. Se il resto r vale 0, l'algoritmo termina ed il risultato è m
4. Altrimenti si ponga $n \leftarrow m$ e $m \leftarrow r$
5. Si torni al punto 2.

Soluzione

Supponiamo di voler calcolare il MCD fra 1638 e 533, utilizzando il suddetto algoritmo. Verranno eseguiti i calcoli visualizzati nella tabella 1.2:

#	n	m	r
01	1638	533	39
02	533	39	26
03	39	26	13
04	26	13	0

Tabella 1.2: Algoritmo di Euclide per il calcolo del MCD

Il MCD fra i numeri 1638 e 533 è quindi 13. Si tratta di un algoritmo di estrema efficienza che permette di calcolare il MCD in pochissimo tempo e senza dover scomporre in numeri primi n e m.

L'algoritmo, inoltre, si presta anche a riflessioni ed analisi supplettive. Cosa succede, ad esempio, se $m > n > 0$? L'allievo testi l'algoritmo nel proposto caso.

1.6 Diagrammi di flusso

Nelle sezioni precedenti si sono fatti due esempi di algoritmi ed essi sono stati esposti in forma descrittiva, utilizzando il linguaggio naturale. Si è, però, accennato a forme differenti di enunciazione e presentazione degli algoritmi, uno dei quali è il **diagramma di flusso**. Si tratta di una rappresentazione grafica che, proprio per tale motivo, è facilmente ed immediatamente comprensibile. Si noti che il diagramma di flusso è solamente un aiuto alla rappresentazione di un algoritmo e non lo descrive in maniera rigorosa, dato che ciascun simbolo, di per sè non identifica un'azione in maniera necessariamente univoca e non ambigua. Rimane, comunque, un valido aiuto per una prima descrizione.

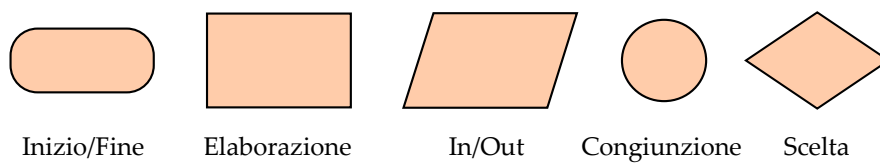



Figura 1.1: Simboli usati nei diagrammi di flusso

Il diagramma di flusso utilizza principalmente i simboli grafici visualizzati in fig. 1.1.

A questi simboli va aggiunto il simbolo di **freccia** ($\rightarrow$), che indica la direzione del flusso d'informazione.

Il simbolo **Inizio/Fine** viene posto, appunto, all'inizio e alla fine dell'algoritmo, ad indicare graficamente rispettivamente il suo punto di ingresso e di uscita. Dal simbolo di Inizio può uscire una sola freccia, come pure nel simbolo di Fine entra una sola freccia. Tipicamente il simbolo di Inizio contiene il nome del processo di cui, appunto, è l'inizio, mentre il simbolo di Fine contiene semplicemente la parola "Fine" (oppure "Return", "Ret", "Ritorna", ecc.). Si noti che se si arriva alla fine del processo da rami (azioni) differenti, è comunque buona norma non terminare tutti i processi direttamente sul simbolo Fine, ma sull'unica freccia che vi accede. Un esempio di quanto detto è proposto in fig. 1.2.

Figura 1.2: Inizio e Fine

Il simbolo di **Elaborazione** sta ad indicare una o più azioni (elaborazioni) che l'algoritmo esegue. Tale simbolo prevede una sola freccia entrante ed una sola freccia uscente. All'interno del blocco viene evidenziato, normalmente in linguaggio naturale, ma il più chiaro e sintetico possibile, l'azione o le azioni che il blocco esegue. La figura 1.3 è un esempio di quanto detto.

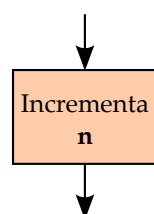



Figura 1.3: Elaborazione

Il simbolo di **In/Out** indica una elaborazione effettuata mediante un dispositivo di *Input* (tipicamente la tastiera, ma non solo) oppure mediante un dispositivo di *Output* (tipicamente il video, oppure una stampante, ma non sono le uniche possibilità). Tale simbolo prevede una sola freccia entrante ed una sola freccia uscente. Anche in questo caso all'interno del blocco si pone una breve indicazione dell'azione eseguita. La fig. 1.4 visualizza un esempio di lettura da tastiera e di scrittura su video.

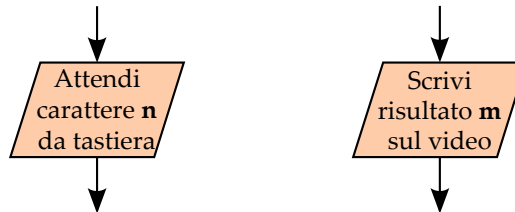



Figura 1.4: In e Out

Il blocco **Scelta** valuta se un determinato predicato è vero o falso. Il predicato è succintamente visualizzato all'interno del simbolo. Si noti che l'eventuale punto di domanda alla fine del predicato trasforma quest'ultimo in domanda. Pur non considerando tale scelta un grave errore si sconsiglia tale abitudine. Nel caso in cui non si intendesse seguire il consiglio appena formulato, si dovrà perlomeno modificare le indicazione poste sulle frecce d'uscita (V: vero; F: falso) in Si e No. Il simbolo prevede una sola freccia entrante e due frecce uscenti. Un esempio di quanto detto è visualizzato in fig. 1.5.

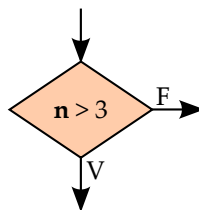



Figura 1.5: Scelta

La **Congiunzione** serve a collegare fra loro due punti di un diagramma di flusso particolarmente complesso e normalmente posti su due fogli distinti. All'interno del simbolo si pone solitamente una lettera maiuscola dell'alfabeto inglese. Due punti che devono essere collegati conterranno nel simbolo di Congiunzione la stessa lettera.

Si consiglia, tuttavia, un uso moderato delle Congiunzioni e di ricorrervi solo quando non è possibile farne a meno. Dover saltare da una pagina all'altra alla ricerca della continuazione di un determinato ramo non contribuisce alla chiarezza del diagramma di flusso, tanto meno se le Congiunzioni sono più d'una e poste su pagine diverse. Una pessima scelta, ad esempio, è quella di avere più di due Congiunzioni con la stessa lettera. Ciò significa che il lettore deve mentalmente congiungere 3-4 punti posti, magari, su pagine differenti con tanti saluti alla chiarezza e all'immediatezza del diagramma di flusso.

Le Congiunzioni possono essere talvolta usate anche all'interno della stessa pagina. In tal caso lo scopo è quello di omettere una freccia dal percorso eccessivamente tortuoso e confuso.

Fin dove è possibile, comunque, si cercherà di evitare tale simbolo. Un esempio di congiunzione è visualizzato in figura 1.6.

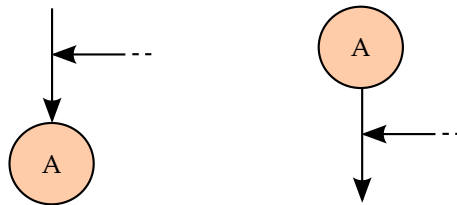



Figura 1.6: Congiunzione

Di seguito, a mo' d'esempio, di come i singoli simboli possano essere usati per rappresentare un algoritmo, è rappresentato il diagramma di flusso dell'algoritmo di Euclide, trattato nella sezione 1.5.

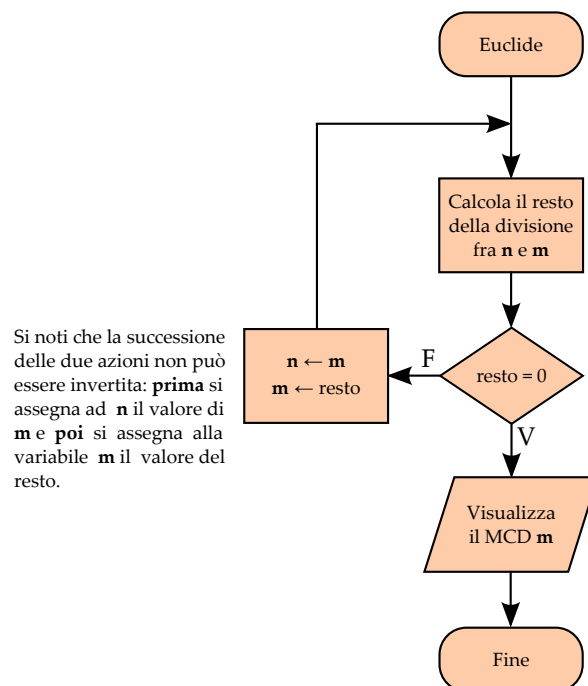



Figura 1.7: Diagramma di flusso dell'algoritmo di Euclide

Osservando il diagramma si rileva che:

- il simbolo di Inizio contiene un nome mnemonico che identifica il diagramma;
- una sola freccia esce dal simbolo Euclide ed una sola freccia entra nel simbolo Fine;

- dai blocchi di Elaborazione e di In/Out entra una sola freccia ed esce una sola freccia;
- dal simbolo Scelta entra una sola freccia e ne escono due;
- ogni blocco contiene una sintetica indicazione dell'azione svolta;
- al fine di evitare ambiguità è stato aggiunto un commento ad un blocco;
- il flusso delle azioni si sviluppa dall'alto verso il basso;
- il *layout* grafico è stato curato in modo da permetterne una leggibilità ottimale;
- non vi sono Congiunzioni, perché graficamente non necessarie.

1.7 Selezione, Sequenza e Iterazione

Un qualsiasi algoritmo è sempre composto da tre strutture fondamentali:

1. Sequenza;
2. Selezione;
3. Iterazione.

Dette strutture sono facilmente identificabili nel diagramma di flusso proposto in fig. 1.7:

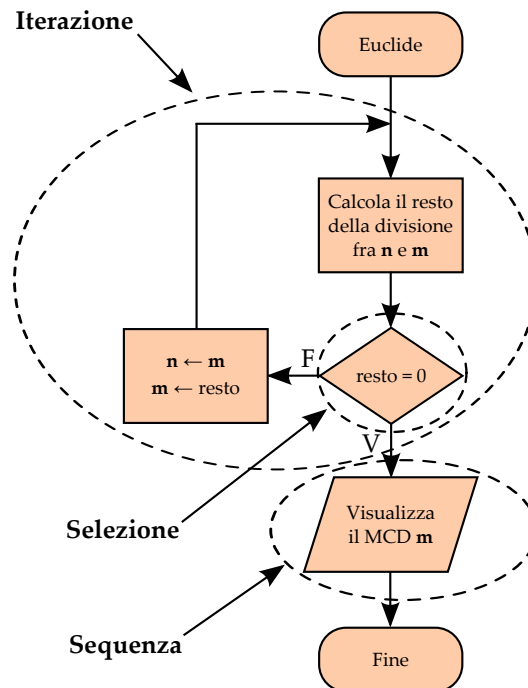



Figura 1.8: Strutture formanti il diagramma di flusso

1.7.1 La Selezione nei diagrammi di flusso

La Selezione corrisponde al simbolo della Scelta e permette, appunto, di operare una scelta a seconda del valore logico di un determinato predicato. Nel presente caso viene valutato il predicato `resto = 0`, ovvero si valuta se il resto della precedente divisione intera era risultato pari a zero. Se detto predicato è vero (V) si visualizza il risultato, che risiede in **m**, altrimenti (F) si esegue un'ulteriore divisione.

In linea generale la Selezione permette di **prendere una decisione**. Esistono sostanzialmente due tipi di Selezione, visualizzate entrambe in fig. 1.9. Nel

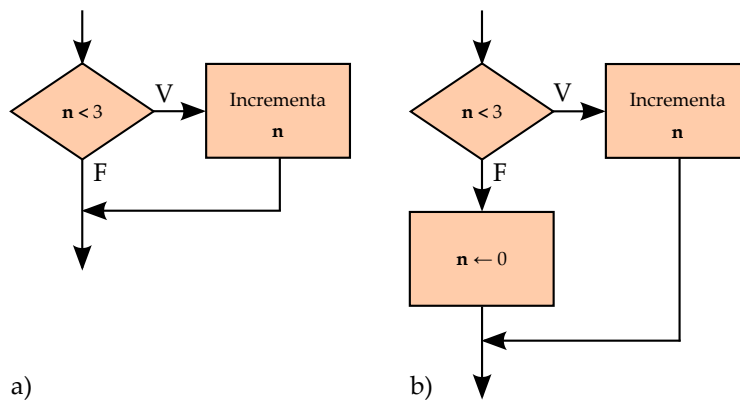



Figura 1.9: Tipi di Selezione

primo caso (fig. 1.9a) il blocco di Scelta valuta se **n**<3: se il predicato è vero, **n** viene incrementato, altrimenti non viene eseguita alcuna azione. Dal punto di vista linguistico tale situazione è equivalente all'espressione "*se ... allora ...*", dove i puntini di sospensione posti dopo il *se* corrispondono al predicato e quelli posti dopo l'*allora* all'azione conseguente. Si noti che talvolta può essere conveniente eseguire l'azione quando il predicato è vero e talvolta quando è falso.

Nel secondo caso (fig. 1.9b) si hanno, invece, due distinte azioni: una che viene eseguita quando il predicato è falso (viene azzerata **n**) ed una quando il predicato è vero (**n** viene incrementata). Dal punto di vista linguistico tale situazione è equivalente all'espressione "*se ... allora ... altrimenti ...*". Nel caso rappresentato in figura, i puntini di sospensione posti dopo l'*allora* corrispondono all'incremento di **n** e quelli posti dopo l'*altrimenti* all'azzeramento di **n**.

La scelta dell'utilizzo del primo o del secondo tipo di selezione dipende dal contesto.

1.7.2 La Sequenza nei diagrammi di flusso

La Sequenza corrisponde a uno o più blocchi di Elaborazione, In/Out e/o Congiunzione. Nel caso del diagramma di flusso di fig. 1.8 una possibile Sequenza

è rappresentata dalla visualizzazione del risultato. Altre Sequenze sono rappresentate dal blocco in cui si calcola il resto della divisione fra **n** e **m** e dal blocco in cui **n** ed **m** sono aggiornate.

Come già fatto notare attraverso il commento posto in fig. 1.7 la successione in cui le due azioni (**n** ← **m** e **n** ← resto) nel caso specifico non può essere invertita, quindi diverse azioni poste all'interno di un blocco di Elaborazione sono da intendersi eseguite cronologicamente dall'alto verso il basso.

In linea generale la Sequenza permette **l'esecuzione ininterrotta e completa di una serie di azioni**. Si sottolinea il fatto che la Sequenza deve essere "ininterrotta e completa". Essa può essere infatti interrotta da una Selezione (oppure dalla semplice fine dell'algoritmo) o essere eseguita parzialmente. In quest'ultimo caso parte della Sequenza apparterrebbe in realtà ad una Selezione o ad una Iterazione. Si esamini, per maggior chiarezza, la figura 1.10.

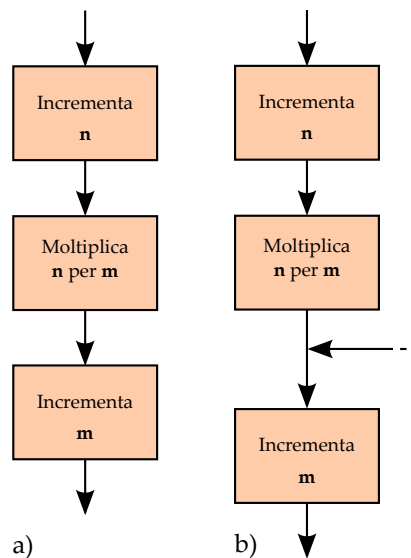



Figura 1.10: Sequenza e falsa Sequenza

In fig. 1.10a è mostrata una corretta Sequenza. Tre azioni (Incrementa **n**, Moltiplica **n** per **m** e Incrementa **m**) sono eseguite ininterrottamente e completamente. In fig. 1.10b, invece, è mostrata una Sequenza che in realtà non è tale: fra l'ultimo ed il penultimo blocco è posta una freccia (proveniente da altra parte dell'ipotetico diagramma di flusso) che lascia intendere che l'ultimo blocco può essere eseguito, in certe situazioni, senza che siano stati eseguiti i due blocchi precedenti. Ciò significa che i due blocchi superiori appartengono in realtà ad una Selezione, oppure il blocco inferiore appartiene ad un'Iterazione.

1.7.3 L'Iterazione nei diagrammi di flusso

L'Iterazione è solitamente formata da più di un blocco. Si tratta, quindi, della struttura più complessa. E' sempre composta almeno da una Scelta e, nor-

malmente, da almeno un blocco di Elaborazione. Nel diagramma di flusso di fig. 1.8 l'Iterazione evidenziata comprende il blocco di Scelta e due blocchi di Elaborazione: quello nel quale viene eseguita la divisione intera e quello nel quale vengono aggiornate le variabili **n** ed **m**.

L'Iterazione permette **l'esecuzione ciclica di una Sequenza** (anche nulla). Due caratteristiche identificano senza ambiguità l'Iterazione: il tipo di scelta (iniziale o finale) e il numero di cicli (definito o non definito). Da tale punto di vista si possono catalogare tre tipi di Iterazione:

- Iterazione con numero di cicli definito (e scelta iniziale);
- Iterazione con numero di cicli non definito e scelta iniziale;
- Iterazione con numero di cicli non definito e scelta finale.

L'Iterazione con numero di cicli definito permette di ripetere una determinata Sequenza un numero di volte definito a priori, ossia noto prima di eseguire per la prima volta detta Sequenza. Siccome tale numero può valere anche zero, questo tipo di Iterazione è sempre a scelta iniziale.

Questo tipo di Iterazione necessita di un **contatore** o **indice** che viene inizialmente caricato con il numero di ripetizioni da eseguire. L'indice viene poi ciclicamente decrementato dopo ogni esecuzione della Sequenza. In fig. 8.1 è mostrato un esempio di tale tipo di Iterazione. La prima azione che deve es-

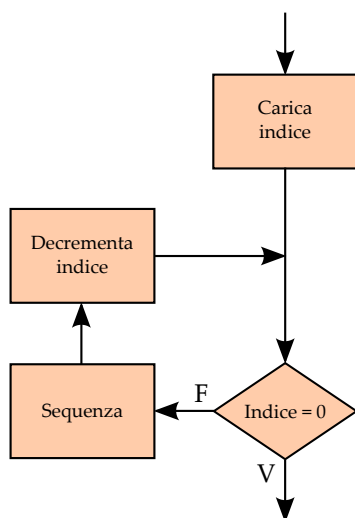



Figura 1.11: Iterazione con numero di cicli definito

sere eseguita consiste nel caricamento dell'indice con il numero di iterazioni, ovvero il numero di volte che la Sequenza deve essere eseguita. L'azione immediatamente seguente è una Scelta che valuta il valore dell'indice. Se vale 0 la Sequenza non viene eseguita, altrimenti si. Dopo che la Sequenza è stata eseguita si deve decrementare l'indice e valutare nuovamente il suo valore. Tale ciclo continua finché l'indice vale 0.

Si noti che se l'indice viene inizialmente caricato con il valore 0, la Sequenza non viene mai eseguita. La presente Iterazione è detta a Scelta iniziale (o a test iniziale), perché la Scelta avviene prima della esecuzione della Sequenza. Si noti anche che non è corretto operare altre modifiche all'indice fuorché il suo

naturale decremento alla fine della Sequenza, perché ciò renderebbe incoerente il numero di cicli inizialmente stabilito con quello effettivamente eseguito.

Infine, l'Iterazione con numero di ciclo definito va utilizzato ogni qualvolta è noto a priori il numero di volte che la Sequenza deve essere ripetuta. Usare detta Iterazione in altre occasioni o modificare l'indice per accelerare o ritardare l'uscita è sempre errato dal punto di vista logico. Se ciò si rende necessario significa che si è scelta l'Iterazione errata.

L'Iterazione con numero di cicli non definito e scelta iniziale permette di ripetere una determinata Sequenza finché non si avvera una certa condizione logica. La particolarità di questa Iterazione è che non si conosce a priori il numero delle ripetizioni e che in determinate condizioni è possibile non eseguire mai la Sequenza.

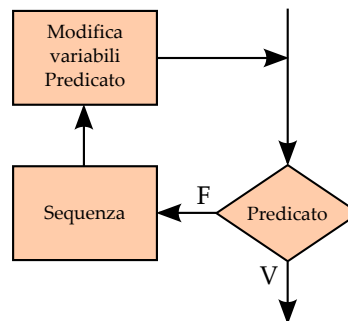



Figura 1.12: Iterazione con numero di cicli non definito e Scelta iniziale

Per prima cosa viene valutato il Predicato e se esso è vero [1] si esce direttamente dall'Iterazione senza eseguire mai, nemmeno una volta, la Sequenza. Se, invece, il Predicato è falso la Sequenza viene eseguita.

E' importante sottolineare che durante l'esecuzione della Sequenza **deve essere modificato il valore di almeno una delle variabile che formano il Predicato**. Se ciò non avvenisse, il Predicato non si modificherebbe mai e l'Iterazione verrebbe eseguita all'infinito. Nel diagramma di fig. 8.12 tale necessità è evidenziata separandola dalla Sequenza vera e propria.

Questa Iterazione va utilizzata ogni qualvolta si verificano le seguenti due condizioni:

- non è noto a priori il numero delle ripetizioni della Sequenza;
- vi è la possibilità di non dover mai eseguire la Sequenza.

La prima condizione si ha quando non è possibile (o è inutilmente complicato) determinare per via analitica il numero delle ripetizioni della Sequenza, oppure quando dipende da fattori esterni (ad esempio, umani). La seconda condizione può essere anche del tutto teorica o probabilisticamente remota, ma se esiste la possibilità che la Sequenza non venga mai eseguita (in concomitanza di un numero di cicli non definito a priori) allora l'Iterazione da usare è la presente.

[1]Le "uscite" V ed F dal simbolo Scelta sono in linea di principio sempre intercambiabili, in qualsiasi tipo di Iterazione o Selezione. Non esiste, quindi, una regola che imponga l'uscita dall'Iterazione quando il Predicato è vero oppure falso.

Iterazione con numero di cicli non definito e scelta finale permette anch'essa di ripetere una determinata Sequenza finché non si avvera una certa condizione logica (come per il caso precedente). La particolarità di questa Iterazione è che non si conosce a priori il numero delle ripetizioni e che la Sequenza viene sempre eseguita almeno una volta. Il sottostante diagramma è un esempio di quanto detto.

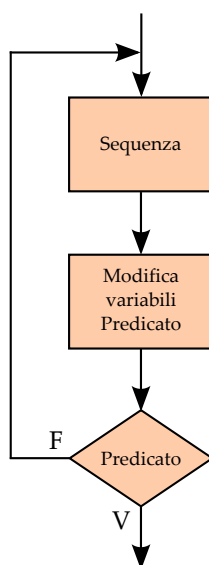



Figura 1.13: Iterazione con numero di cicli non definito e Scelta finale

Anche in questo caso la Sequenza è tenuta distinta dalla modifica delle variabili del Predicato, anche se possono (e talvolta devono) benissimo coesistere. Si nota che la Sequenza deve essere eseguita almeno una volta, dato che il blocco di Scelta è posto in fondo. Anche in questo caso è importante sottolineare che la modifica delle variabili del predicato è fondamentale affinché l'Iterazione non si trasformi in un ciclo infinito [2].

Questa Iterazione va utilizzata ogni qualvolta si verificano le seguenti due condizioni:

- non è noto a priori il numero delle ripetizioni della Sequenza;
- la Sequenza deve sempre essere eseguita almeno una volta.

Un esempio classico della presenza delle suddette due condizioni è la digitazione da tastiera di un numero compreso fra un limite inferiore ed un limite superiore. Se il numero digitato è fuori dai limiti indicati, la richiesta di immissione va ripetuta. Siccome non è dato sapere quante volte l'utente sbaglierà a digitare il numero e siccome almeno una volta il numero deve essere digitato, per poterlo verificare, questa Iterazione si presta alla perfezione per risolvere il problema.

[2] Si noti che in tutte le applicazioni *consumer* basate su microprocessore o microcontrollore esiste sempre un ciclo infinito che "contiene " il programma applicativo. In tal caso il Predicato non viene mai modificato.

1.8 La *subroutine*

Un concetto molto utile quando si deve documentare un algoritmo è quello di *subroutine* (o sottoprogramma in una traduzione non impeccabile). Nella sostanza si tratta di una Sequenza poco dettagliata che è documentata altrove nei particolari. Pur potendo, quindi, utilizzare a tale scopo il simbolo di Elaborazione, aluni testi usano il simbolo visualizzato in fig. 1.14 per documentare la subroutine.

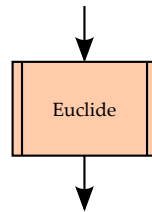



Figura 1.14: Simbolo di Subroutine

La subroutine viene utilizzata per dettagliare separatamente (solitamente su altri fogli) parti di un diagramma di flusso volutamente poco particolareggiato ed usato ripetutamente. Un esempio di uso di subroutine è illustrato in fig. 1.15

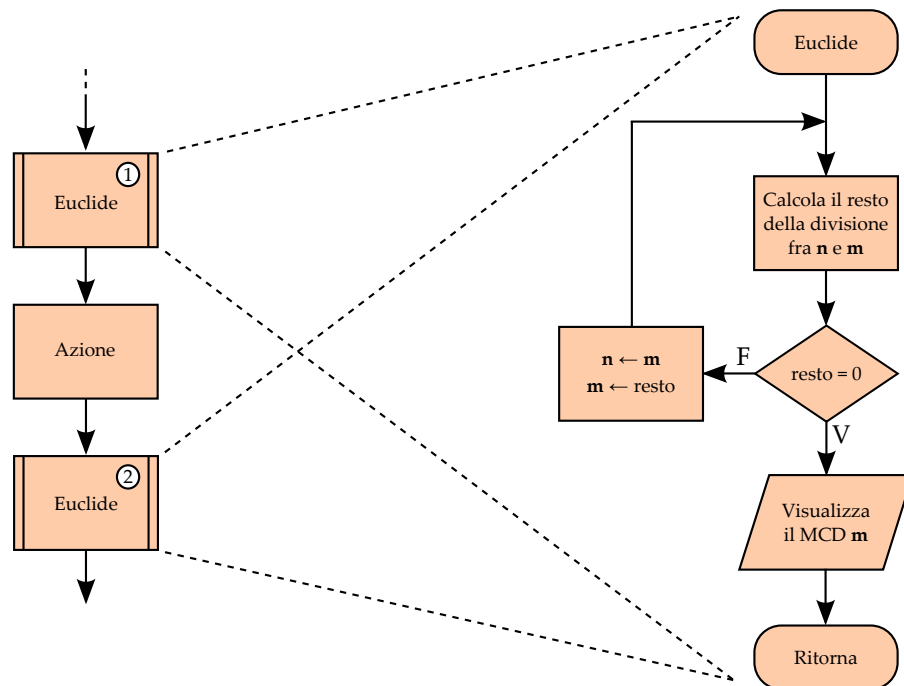



Figura 1.15: Esempio di Subroutine

La parte sinistra della figura mostra una Sequenza che richiama due volte il sottoprogramma Euclide: una prima volta prima di Azione ed una seconda

volta subito dopo Azione.

Quando nel diagramma di flusso principale (la Sequenza) viene richiamato per la prima volta il sottoprogramma Euclide, l'attenzione si sposta sul diagramma di flusso che documenta Euclide. In tale diagramma di flusso l'algoritmo di Euclide è documentato nei dettagli ed è possibile, se lo si desidera, capire come l'algoritmo funziona. Se, invece, non si ritiene necessario dedicare attenzione al dettaglio dell'algoritmo ci si può concentrare sulla Sequenza senza essere distratti dai dettagli. Comunque, alla fine del sottoprogramma si ritorna esattamente alla fine del blocco chiamante, ovvero alla fine del blocco 1.

Dopo aver eseguito il blocco Azione, viene richiamatom per la seconda volta il sottoprogramma Euclide. Idealmente l'attenzione si sposta, se necessario, al diagramma di flusso che documenta Euclide, al termine del quale si ritorna esattamente sotto il blocco chiamante, in questo caso alla fine del blocco 2.

Questo modo di presentare la documentazione, soprattutto se corposa, è altamente consigliabile, per almeno due motivi:

- invoglia l'autore a pensare al problema in maniera strutturata e a suddividerlo logicamente in Sequenze;
- permette al lettore del diagramma di flusso di decidere il grado di approfondimento della lettura della documentazione.

Si consiglia, quindi, un uso intensivo delle subroutines sia al fine di semplificare organicamente il problema, suddividendolo in sottoproblemi affrontabili separatamente, sia al fine di una documentazione più leggibile ed organizzata.

1.9 Il buon diagramma di flusso

Un buon diagramma di flusso non deve essere soltanto efficiente e corretto, ma deve soddisfare anche esigenze di leggibilità, visto che il suo compito è quello di fornire una immediata comprensione dell'algoritmo che documenta. In fig. 1.16 è rappresentato un brutto diagramma di flusso.

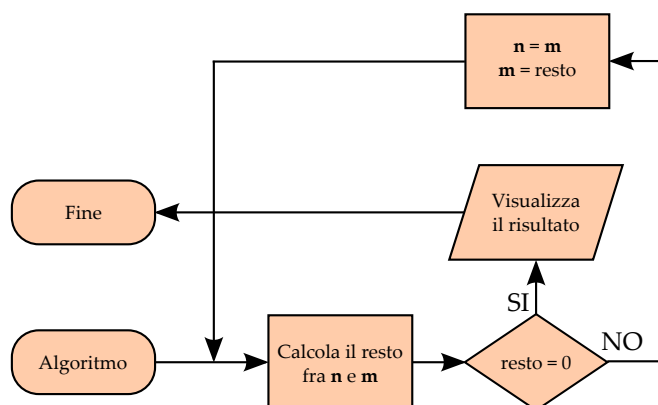



Figura 1.16: Un brutto diagramma di flusso

Esso incorpora molti difetti da evitare:

- l'algoritmo non si sviluppa prevalentemente dall'alto verso il basso;
- il simbolo di Inizio non esplica in alcun modo la funzione dell'algoritmo;
- la leggibilità del diagramma è molto bassa;
- le azioni nei blocchi sono indicate in maniera poco precisa (non si capisce, ad esempio, quale sia il risultato);
- non vi sono commenti esplicativi che evitino ambiguità;
- il simbolo '=' è usato sia per indicare uguaglianza che assegnazione;
- le uscite dalla Scelta sono indicate con SI e NO.

Un buon diagramma deve, invece, essere chiaro e facilmente leggibile. E' necessario curare sia l'aspetto grafico (evitando disposizioni confuse, incroci di frecce, ecc.) che quello testuale (evitando esposizioni grossolane e/o superficiali, simbologie incoerenti, ecc.).

Un buon esercizio potrebbe consistere nel confrontare il diagramma di flusso di fig. 1.16 con quello di fig. 1.7 cercando tutti i difetti dell'uno ed i pregi dell'altro.

Infine, un'ultima riflessione.

E' possibile descrivere degli algoritmi in maniera chiarissima, che soddisfano perfettamente le 5 caratteristiche illustrate nella sezione 1.1, che illustrano un processo efficiente e che sono di immediata comprensione, ma che non sono perfettamente coerenti con le tre strutture fondamentali illustrate nella sezione 1.7 (Sequenza, Selezione e Iterazione).

Ciò si rende possibile se si violano le strutture di Selezione e/o di Iterazione. Quando si sono presentate dette strutture (vedi figg. 1.9a, 1.9b, 8.1, 8.12 e 8.6) si è tacitamente supposto che esse non fossero modificabili, ovvero che vi fosse una sola freccia entrante dalla struttura ed una sola freccia uscente. Violando, però, dette strutture è possibile tracciare un diagramma di flusso simile a quello rappresentato in fig. 1.17. Una tale struttura non è prevista fra

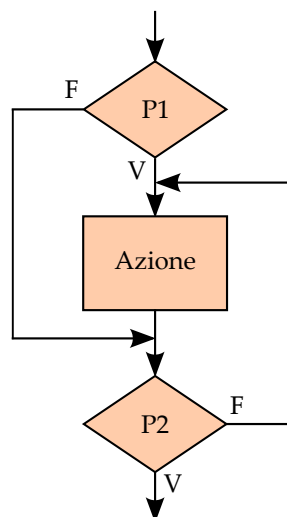



Figura 1.17: Struttura da evitare

quelle fondamentali ma può esservi agevolmente ricondotta, come illustrato in fig. 1.18.

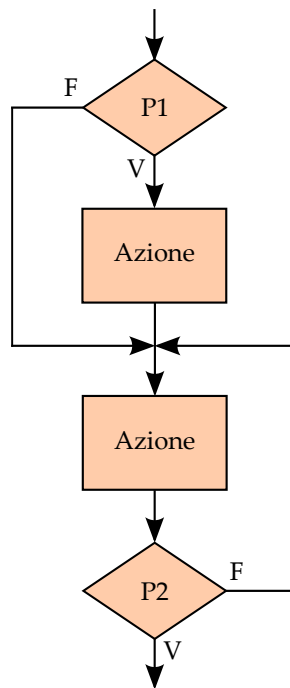



Figura 1.18: Struttura corretta

Allenare la mente al rispetto delle tre strutture fondamentali favorisce la successiva comprensione delle medesime quando verranno presentate secondo la sintassi del linguaggio di programmazione C. Non solo: dette strutture aiutano a suddividere i problemi complessi in sottoproblemi di difficoltà minore, evitando contorsionismi mentali e scomposizioni che aumentano la difficoltà di comprensione dell'algoritmo anziché diminuirla. Si sconsiglia, pertanto, di derogare dalle strutture fondamentali, nonostante vi sia modo di elencare esempi virtuosi di violazione. Detti esempi sono però riconducibili ad una programmazione piuttosto avanzata, che non verrà trattata nelle presenti pagine.

1.10 La pseudocodifica

I diagrammi di flusso non sono l'unico modo per rappresentare un algoritmo. Un metodo molto diffuso, molto pratico e veloce, anche se forse meno immediato ed efficace da capire è rappresentato dalla pseudocodifica. Essa utilizza il linguaggio naturale, qualche regola e delle **parole chiave** per riprodurre le stesse strutture fondamentali dei diagrammi di flusso: la Selezione, la Sequenza e l'Iterazione.

Le parole chiave che sono utilizzate normalmente nell'uso della pseudocodifica sono le seguenti:

Parola chiave
INIZIO
FINE
SE
ALLORA
ALTRIMENTI
RIPETE
FINCHE'

Tabella 1.3: Parole chiave usate nella pseudocodifica

Dette parole si possono anche modificare, aumentare o ridurre, purchè non venga meno l'immediatezza della comprensione. Normalmente esse sono scritte in maniera evidenziata (colore diverso dal normale testo, a tutte maiuscole, in grassetto, ecc.).

Al fine di aumentare la leggibilità dell'algoritmo ed evidenziare le diverse strutture, si utilizzano alcune semplici regole, fra cui l'**indentazione** (che consiste in uno spostamento verso destra della struttura da scrivere) e la scrittura su una riga a sè stante delle singole parti componenti la struttura. Gli esempi che seguono illustrano anche graficamente quanto detto.

1.10.1 La Selezione nella pseudocodifica

In fig. 1.19 è evidenziato un esempio di Selezione scritto in pseudocodifica.

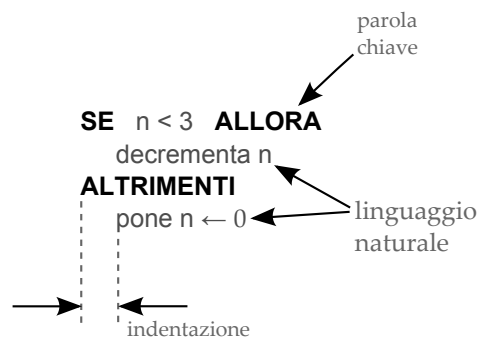



Figura 1.19: Esempio di Selezione

In essa si distinguono le parole chiave scritte in maiuscolo grassetto (**SE**, **ALLORA** e **ALTRIMENTI**) che compongono sostanzialmente la struttura della Selezione. Si noti che esse riproducono la stessa sintassi che verrebbe utilizzata nel linguaggio naturale. Il fatto che siano evidenziate serve solo a circoscrivere in maniera non ambigua la struttura e avvicinare quanto più possibile (senza invaderne il campo) il linguaggio naturale al linguaggio formale.

L'indentazione permette di porre in evidenza le singole parti della Selezione, ovvero la Sequenza che viene eseguita se il Predicato è vero e quella che viene eseguita se il Predicato è falso. Quanto ciò sia estremamente importante appare evidente nel seguente esempio.

Si supponga di dover documentare mediante pseudocodifica l'algoritmo di individuazione dell'anno bisestile. In linguaggio naturale esso potrebbe essere esposto in forma semplificata nel seguente modo[3]:

Un anno solare è da considerarsi bisestile se è divisibile per 4 ma non per 100, a meno che non sia divisibile per 400.

Un tale modo di descrivere un algoritmo si presta a interpretazioni errate o perlomeno difficoltose. L'allievo può tentare, a mo' d'esercizio, di interpretare l'algoritmo e valutare la difficoltà di interpretazione.

Una descrizione mediante pseudocodifica permette di esporre il concetto con maggiore chiarezza:

Listing 1.1: Anno bisestile

```
SE anno divisibile per 4 ALLORA
   SE anno divisibile per 100 ALLORA
      SE anno divisibile per 400 ALLORA
         l'anno e' bisestile
      ALTRIMENTI
         l'anno non e' bisestile
   ALTRIMENTI
      l'anno e' bisestile
ALTRIMENTI
   l'anno non e' bisestile
```

Il listato 1.1 evidenzia mediante l'indentazione le singole strutture e rende più facile l'interpretazione dell'algoritmo. In fig. 1.20 la pseudocodifica è analizzata evidenziando le tre strutture di cui è composta.

```
Struttura 1 { SE anno divisibile per 4 ALLORA
  Struttura 2 { SE anno divisibile per 100 ALLORA
    Struttura 3 { SE anno divisibile per 400 ALLORA
                     l'anno e' bisestile
                  ALTRIMENTI
                     l'anno non e' bisestile
                ALTRIMENTI
                   l'anno e' bisestile
              ALTRIMENTI
                 l'anno non e' bisestile
```

Figura 1.20: Analisi dell'algoritmo dell'anno bisestile

Si valuti la struttura più interna (Struttura 3): se l'anno è divisibile per 400 è sicuramente un anno bisestile; se non è divisibile per 400 rimane comunque divisibile per 100 (altrimenti non si sarebbe arrivati a valutare la Struttura 3), quindi non è bisestile.

Si valuti la Struttura 2: il caso in cui l'anno sia divisibile per 100 è già stato esaminato; se non è divisibile per 100, rimane comunque divisibile per 4

[3]Semplificata perchè in realtà bisognerebbe valutare anche la divisibilità per 4000

(altrimenti non si sarebbe arrivati a valutare la Struttura 2), quindi l'anno è bisestile.

Infine la struttura più esterna (Struttura 1): il caso in cui l'anno sia divisibile per 4 è già stato esaminato; se non è divisibile per 4 l'anno non è bisestile.

Poter facilmente identificare una struttura (traduzione: abbinare correttamente l'**ALTRIMENTI** al **SE**) facilita la corretta interpretazione della stessa.

La Selezione può essere anche priva dell'**ALTRIMENTI**, nel qual caso un esempio di pseudocodifica è il seguente:

Listing 1.2: Esempio di Selezione del tipo SE .. ALLORA

```
SE anno divisibile per 400 ALLORA
   l'anno e' bisestile
```

Siccome non esiste un simbolo standard che indichi l'inizio di un commento, se si vuole aggiungere un testo che commenti la pseudocodifica è possibile farlo nel seguente modo:

Listing 1.3: Esempio di pseudocodifica commentata

```
Commento: Le parti ambigue vanno sempre commentate, al
fine di aggiungere informazioni supplettive (anche ri-
dondanti) che aiutano ad interpretare la codifica. Fi-
ne commento.

SE anno divisibile per 400 ALLORA
   l'anno e' bisestile
```

1.10.2 La Sequenza nella pseudocodifica

In fig. 1.21 è evidenziato un esempio di Sequenza scritto in pseudocodifica.

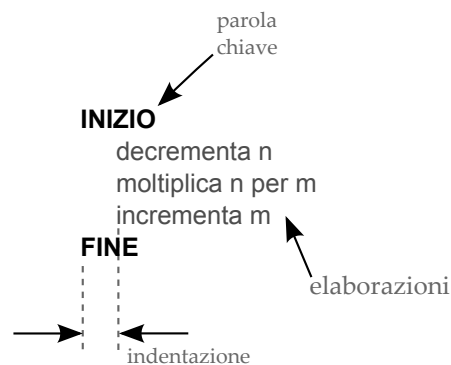



Figura 1.21: Esempio di Sequenza

Le parole chiave **INIZIO** e **FINE** delimitano la Sequenza che, nel presente caso, è formata da tre distinte elaborazioni (`decrementa n`, `moltiplica n per m`, `incrementa m`). Dette parole chiave sono sempre necessarie quando si vogliono racchiudere diverse elaborazioni in un'unica Sequenza. Anche in

questo caso l'indentazione aiuta graficamente e porre in risalto la Sequenza che si vuole indicare.

Tecnicamente sarebbe possibile porre dei **punti di entrata** nella Sequenza. Si è già visto come questa procedura spezzi la Sequenza in due parti appartenenti a due strutture fondamentali differenti. Non si ritiene quindi utile indugiare sull'argomento, offrendo allo studente cattivi abiti mentali.

1.10.3 L'Iterazione nella pseudocodifica

Anche nel caso della pseudocodifica l'Iterazione è la struttura più complessa e si possono catalogare i tre tipi distinti di Iterazioni descritti nella sezione 1.7.3.

Un esempio di **Iterazione con numero di ciclo definito** è graficamente descritto in fig. 1.22. Questo tipo di Iterazione è caratterizzato dal fatto che il

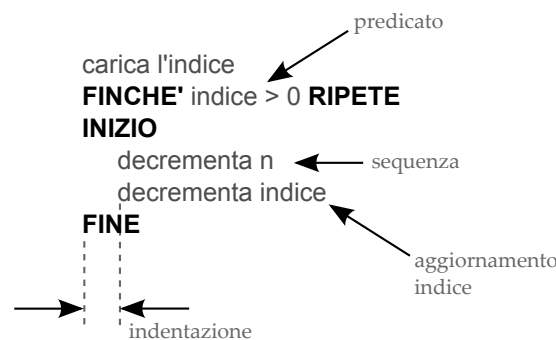



Figura 1.22: Pseudocodifica di Iterazione con numero di cicli definito

numero di ripetizioni è noto a prima dell'esecuzione della Sequenza e dal fatto che la Scelta è posta all'inizio del ciclo. Il Predicato relativo alla Scelta verifica, come peraltro visto a proposito dei diagrammi di flusso, se l'indice (o contatore) è maggiore di zero oppure no: se è maggiore di zero entra nel ciclo ed esegue la Sequenza, altrimenti esce dal ciclo senza eseguirla.

Si noti che l'indice viene aggiornato dopo che la Sequenza è stata eseguita[4], in modo che non vi sia ambiguità sullo stato dell'indice all'interno della medesima.

Anche in questo caso si ribadisce l'importanza di non modificare (se non durante il decremento) in alcun modo l'indice del ciclo al fine di forzare l'uscita o prolungare la permanenza nell'Iterazione. Se ciò dovesse rendersi necessario si dovrebbe ripensare la soluzione adottata e optare, probabilmente, per un'Iterazione a numero di cicli non definito. Si insiste molto su tale aspetto perché solitamente non si dedica sufficiente tempo e attenzione alla scelta del

[4]Tale caratteristica è legata all'Iterazione con numero di cicli definito e Scelta iniziale, indipendentemente dal tipo di rappresentazione (diagrammi di flusso o pseudocodifica). Questo perché sovente la Sequenza utilizza l'indice al proprio interno e deve quindi essere noto quando esso viene aggiornato. Per convenzione viene aggiornato alla fine della Sequenza.

tipo di Iterazione, utilizzando spesso sempre l'unica Iterazione che si è imparato ad usare e che si conosce meglio. Ogni situazione, invece, richiede un ben determinato tipo di Iterazione e non altri.

L'**Iterazione a numero di cicli non definito e Scelta iniziale** è, invece, illustrata in fig. 1.23.

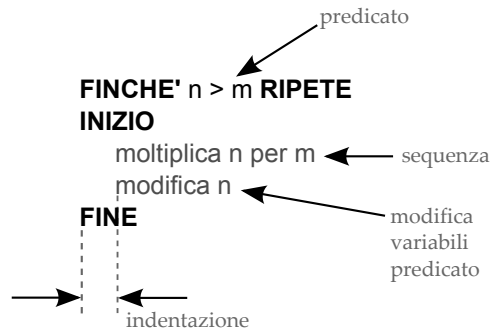



Figura 1.23: Pseudocodifica di Iterazione con numero di cicli non definito e Scelta iniziale

Come già visto, la particolarità di questa Iterazione è data dal fatto che la Sequenza potrebbe non essere mai eseguita. Si tratta probabilmente del più flessibile dei tre cicli, essendo quello che, in assenza di formalismo, potrebbe candidarsi con maggior fortuna a "Iterazione unica", sostituendo i restanti due. Si sottolinea, però, che una tale visione delle Iterazioni non è formalmente corretta e va scoraggiata.

Anche in questo caso è importante sottolineare che all'interno del ciclo almeno una delle variabili del Predicato deve essere modificata, anche se dipende dal contesto se tale modifica vada effettuata prima, dopo o durante la Sequenza.

Si ribadisce che in assenza di modifica delle variabili del Predicato, non vi è alcun modo di uscire dall'Iterazione e, solitamente, ciò costituisce un errore.

L'**Iterazione a numero di cicli non definito e Scelta finale** è l'ultima delle tre ed è illustrata in fig. 1.24.

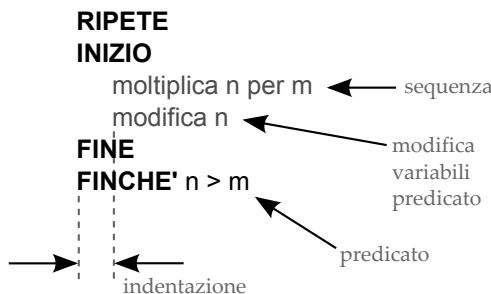



Figura 1.24: Pseudocodifica di Iterazione con numero di cicli non definito e Scelta finale

La particolarità di questo tipo di Iterazione è che la Sequenza viene eseguita sempre almeno una volta, dato che si tratta di un ciclo a test finale. Per quanto riguarda, invece, la modifica delle varibili del Predicato ed il numero delle ripetizioni della Sequenza, vale quanto già detto in precedenza.

1.11 Esercizi svolti

Gli esercizi svolti riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 1. Essi sono presentati, più o meno, in ordine di difficoltà crescente, in modo da accompagnare lo studente attraverso il difficile compito di maturazione della padronanza dell'uso dei diagrammi di flusso. Si cercherà di dare un metodo di risoluzione allo studente attraverso l'esecuzione di detti esercizi.

Esercizio - ◊◊◊ Segno del numero

Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:

1. digitare un numero intero sulla tastiera del PC;
2. valutare se il numero è positivo oppure negativo;
3. stampare a video se il numero è positivo o negativo.

Soluzione

La prima cosa che lo studente deve imparare a fare consiste nel comprendere bene e a fondo il problema. Dedicare qualche minuto alla comprensione delle consegne non è mai tempo perso.

La prima domanda che lo studente deve porsi è: "Cosa mi viene chiesto?" Il testo chiede *"la stesura del diagramma di flusso ..."*, il che significa che il prodotto finale **deve** essere un diagramma di flusso. Esso deve descrivere delle azioni che sono esposte al lettore in linguaggio naturale. **Attenzione**: un algoritmo descritto in linguaggio naturale, ambiguo per definizione, nasconde sempre delle insidie.

Il primo punto delle consegne descrive l'azione di digitare un numero intero sulla tastiera di un PC. Tale azione dovrebbe richiamare alla mente dell'allievo il blocco di **Input**: esso viene utilizzato per definire i *dati di ingresso*. Si desidera spendere qualche parola a tal proposito.

A volte lo studente incontra delle difficoltà nell'astrarre tale concetto. L'azione descritta al punto 1 indica un input di dati. Ciò significa che si deve immaginare che un ipotetico utente digiti un qualsiasi numero intero sulla tastiera e che questo numero venga acquisito dal PC (ossia, nel nostro caso, dall'algoritmo), senza, però, che sia noto. Il numero è stato acquisito, ma non ne conosciamo il valore. E' come se mettessimo la mano nel sacchettino dei numeri della tombola e ne estraessimo uno, *senza, però, leggerlo*: il numero è stato estratto e lo teniamo stretto in pugno, ma non ne conosciamo il valore.

Abbiamo, però, la necessità di riferirci al numero estratto: dobbiamo parlarne, fare dei calcoli con esso, incrementarlo o decrementarlo, o fare chissà cos'altro. A tal fine potremmo riferirci ad esso come al "numerocheabbiamoestrattodalsacchettinoeteniamoinpugnosenzasaperechevaloreha", ma tale definizione risulta scomoda e dispendiosa. Conviene trattarlo come una **variabile**, ad esempio la variabile n.

Ma cosa intendiamo per variabile? A che ambito dobbiamo riferirci? Informativo? Matematico?

In fin dei conti un algoritmo, si è detto a inizio capitolo, potrebbe essere anche la descrizione di una ricetta di cucina. In un'ottica generale, potremmo quindi tentare di definire la variabile come la "rappresentazione di un elemento di un insieme".

La nostra variabile n, quindi, *rappresenta* un *elemento* dell'insieme dei numeri relativi $\mathbb{Z}$.

Continuando nel commento ragionato del testo del problema (l'allievo farebbe bene a commentare criticamente sempre una qualsiasi consegna!), il punto 2 chiede di "*valutare se il numero* (ossia la nostra variabile n) *è positivo oppure negativo*.

Molti allievi potrebbero sostenere che questo punto non presenta grossi problemi. Lo studente pignolo, però, potrebbe chiedersi cosa significhi, in concreto e in questo contesto, *valutare* qualcosa. "Valutare", in questo caso, è un'azione *antecedente* che si concretizza in un'azione *conseguente*. Esempio: "Valuto se piove, prima di prendere l'ombrello". L'azione di "valutare", quindi, non è fine a se stessa, ma costituisce preludio ad un'altra azione. Nel caso del nostro problema orginale, condiziona il tipo di stampa a video.

Il terzo punto del testo è quindi strettamente legato al secondo nel modo già commentato. Tutto sommato il problema potrebbe essere descritto mediante i soli punti 1 e 3, saltando il punto 2.

L'unica precisazione da aggiungere riguarda il termine "stampare". In questo caso assume il significato di "visualizzare". Si tratta quindi di visualizzare a video (azione di **Output**) una frase che specifichi se il numero digitato è positivo oppure negativo.

Tutto chiaro? No, per niente. C'è ancora un punto molto importante da chiarire. Quale? Lo studente tenti di dare una risposta alla domanda senza proseguire nella lettura.

Si auspica che il lettore abbia tentato attivamente e criticamente di individuare cosa rimane da chiarire nel testo. Probabilmente uno studente alle "prime armi" con il *problem solving* e con la lettura critica di un testo, potrebbe essere in difficoltà e sentirsi inadeguato. Bisogna, però, chiarire subito che l'importante non è trovare subito ed in maniera efficente la soluzione, ma cercarla. L'allievo che sta leggendo queste pagine ha probabilmente intrapreso un lungo cammino, non privo di difficoltà, per cui è normale che abbia qualche difficoltà di orientamento.

Il testo del problema è formulato molto male. Si chiede di valutare se il numero digitato è positivo oppure negativo. E se il numero digitato fosse lo zero? Lo zero è l'unico numero reale che non è positivo nè negativo. Si potrebbe sostenere che, appartenendo esso all'insieme dei numeri naturali $\mathbb{N}$, debba per forza essere positivo. Ciò è falso. Normalmente si indica con $\mathbb{N}_0$ l'insieme dei numeri positivi (1, 2, 3, 4, ...) e con $\mathbb{N}$ l'insieme dei numeri non negativi.

Il punto 2 del testo andrebbe, quindi, riformulato chiedendo di "valutare se il numero è positivo, negativo oppure uguale a zero".

Si può ora ragionevolmente supporre di aver letto criticamente e compreso il testo del problema. Si può, quindi, procedere nella stesura del diagramma di flusso.

Ogni diagramma inizia sempre con un unico punto di inizio e finisce, possibilmente, con un unico punto terminale. Non è obbligatorio, ma è buona abitudine dare un nome che ricordi il problema al simbolo di inizio, come in fig. 1.25.

Dal simbolo di Inizio deve uscire una e una sola freccia, in modo che siano chiari ed evidenti sia la direzione che il verso della successione delle azioni che il diagramma illustra.

La prima azione che deve essere evidenziata graficamente è l'input dei dati, ossia il punto 1 del testo. Si supponga che il numero digitato sia rappresentato dalla variabile n[5]. Il diagramma diventa quindi il seguente:

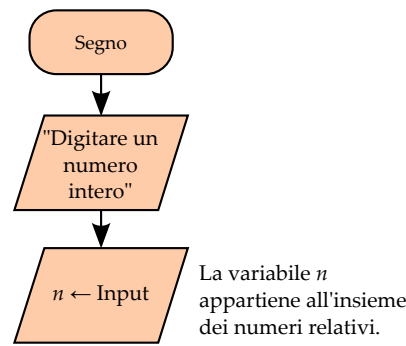



Figura 1.25: Segno: Input dei dati

Si noti che il blocco di Input è sempre preceduto da un blocco di Output che presenti una *frase di cortesia* che introduca l'utente all'azione da svolgere. Nel presente caso la frase dovrebbe invitare l'utente a digitare un numero intero. La frase dovrebbe essere formulata in modo tale da non creare ambiguità e informare chiaramente l'utente di quale debba essere l'azione da compiere.

Il secondo punto descritto nel testo chiede di valutare se il numero è positivo o negativo (oppure uguale a zero). Siccome i casi possibili sono 3 e siccome ogni ogni predicato ha due valori di verità (Vero o Falso), ciò significa valutare il valore di verità di **almeno** due predicati, dato che $2^2 > 3$ (ossia le tre condizioni: +, 0, -).

In realtà è facile dimostrare che per effettuare una scelta multipla fra n possibili casi sono necessari $n - 1$ selezioni[6]. Si conferma, quindi, la necessità di utilizzare due selezioni per ottenere la scelta multipla voluta.

[5] Alcune scuole di pensiero evidenziano anche nei diagrammi di flusso la *dichiarazione delle varibili*. Nelle presenti pagine si è scelto di non farlo, per marcare una certa indipendenza dai linguaggi di programmazione. Dichiarare le variabili anche nei diagrammi di flusso non è, però, una scelta sbagliata, anzi: nell'ottica di un avvicinamento graduale ai linguaggi di programmazione può essere una sana abitudine. Finalità diversa, ma sempre riconducibile alle buone abitudini in ambito di programmazione, ha la *inizializzazione* delle variabili. Anche questa pratica, per i motivi già menzionati, non verrà addottata nelle presenti pagine. Non per questo, però, va scoraggiata.

[6] Lo studente potrebbe provare a dimostrare che per realizzare una scelta multipla di n casi servono $n - 1$ blocchi di selezione.

La valutazione del primo predicato (non ha importanza quale sia: potrebbe essere $n < 0$) potrebbe essere rappresetata mediante il seguente diagramma:

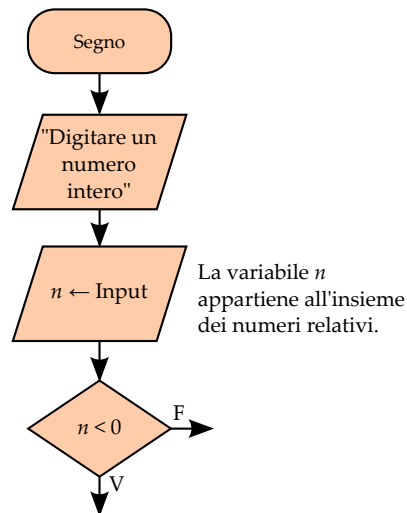



Figura 1.26: Segno: Selezione

Se il predicato è vero, si è già individuato il segno della variabile, altrimenti, se esso è falso, rimane da valutare se il segno sia positivo oppure se la variabile assume il valore zero.

La rappresentazione grafica è data dal diagramma di fig. 1.27.

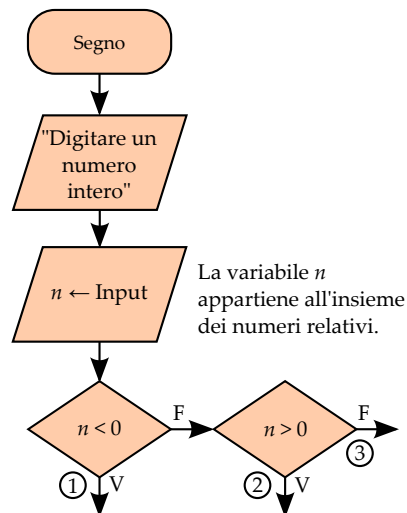



Figura 1.27: Segno: Scelta multipla

Nella figura sono evidenziati (con i numeri 1, 2 e 3) i tre rami "prodotti" dalla scelta multipla: il flusso delle azioni prosegue lungo il ramo 1 se n è negativa; lungo il ramo 2 se n è positiva e lungo il ramo 3 se n vale zero. Questi

tre rami devono confluire in altrettante azioni che evidenzino a video il segno della variabile, come in fig. 1.28:

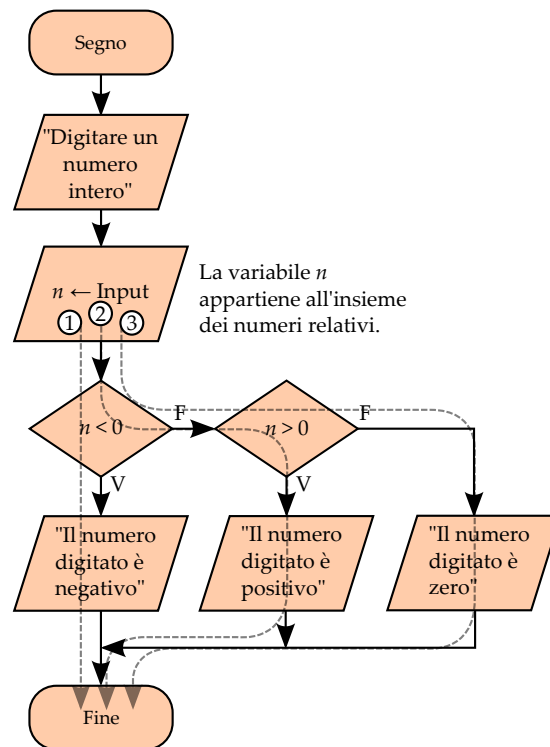



Figura 1.28: Segno: Diagramma finale

I tre blocchi di Output riportano tre frasi diverse che fissano il segno della variabile. Tracciando il diagramma di flusso si è *quasi* terminato l'esercizio. Come ultima azione si deve verificare se ci sono errori evidenti nella sintesi, ovvero nel diagramma. A tal fine conviene "eseguire" l'algoritmo utilizzando degli esempi numerici concreti.

In fig. 1.28 a pagina 32 è anche evidenziato il flusso delle azioni che vengono svolte supponendo i tre casi distinti (caso 1: $n < 0$; caso 2: $n > 0$; caso 3: $n = 0$). La figura evidenzia l'effettiva assenza di errori.

Riassumendo, si sono eseguite quattro azioni fondamentali:

1. si è letto con attenzione il testo. Tutto deve essere chiarissimo prima di continuare con la risoluzione dell'esercizio. Lo studente tenga conto che quando legge criticamente e con attenzione il testo, *sta già risolvendo l'esercizio*. La lettura critica non è un inutile *optional* della risoluzione è **l'inizio** della risoluzione. Cartesio chiamava questo primo passo *evidenza*;
2. la lettura critica ci ha permesso di analizzare il problema e di scomporlo facilmente (era già evidenziato nel testo) in tre parti sostanziali. Questa tecnica di "scomposizione" di un problema complesso in sottoproblemi più facili da affrontare è detta *divide et impera* (dividi e domina). Cartesio chiamava questa seconda fase *analisi*;

3. terminata l'analisi si è potuto tracciare il diagramma di flusso. Questa fase, paradossalmente, è la più facile se le precedenti due sono state eseguite correttamente. Non essere in grado di tracciare il diagramma significa non avere compreso alla perfezione il problema o non averlo analizzato e scomposto adeguatamente. Cartesio chiamava questo terzo passo *sintesi*;
4. infine si è controllato la soluzione alla ricerca di errori e sviste che possono invalidare la soluzione. Tale controllo deve essere effettuato, come al solito, con grande senso critico, senza paura di trovare errori. Cartesio chiamava questo ultimo passo *enumerazione*.

Vengono ora proposti alcuni esercizi simili a quello risolto. Non viene fornita la soluzione ragionata. Lo studente è chiamato a risolverli e a riflettere sulle difficoltà che incontra. Quando si incontra una difficoltà che si ritiene insormontabile (1-2 ore passate senza fare progressi) si deve cercare di identificare meglio possibile l'ostacolo e **descriverlo per iscritto**. Tale scritto deve essere discusso insieme all'insegnante alla prima occasione.

1. ◊◊◊ **Pari o dispari.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) valutare se il numero è pari o dispari;
   (c) stampare a video se il numero è pari o dispari.
2. ◊◊◊ **Divisibile per 3.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) valutare se il numero è divisibile per 3 (ossia senza produrre resto);
   (c) stampare a video se il numero è divisibile per 3 o meno.
3. ◊◊◊ **Minore di 13.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) valutare se il numero è minore di 13;
   (c) stampare a video se il numero è minore di 13 o meno.
4. ◊◊◊ **Maggiore di 5.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) valutare se il numero è maggiore di 5;
   (c) stampare a video se il numero è maggiore di 5 o meno.
5. ◊◊◊ **Compreso fra 10 e 20.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) valutare se il numero è compreso fra 10 e 20 (estremi compresi);
   (c) stampare a video se il numero è compreso fra i limiti dati o meno.

Esercizio - ◊◊◊ Numero decimale

Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:

1. digitare un numero non negativo[7] qualsiasi sulla tastiera del PC;
2. valutare, utilizzando solo le quattro operazioni aritmetiche, se il numero è intero oppure decimale;
3. stampare a video se il numero è intero o decimale.

Soluzione

Apparentemente il problema sembra banale, ma esso nasconde una difficoltà. Non si sta chiedendo se il numero contenga una virgola (o, all'anglosassone, un punto decimale) o meno, ma come sia possibile stabilirlo attraverso le sole quattro operazioni aritmetiche. Si deve, cioè, stabilire come sia possibile, aritmeticamente, distinguere, ad esempio, il numero 12 (intero) dal numero 23.7 (decimale), oppure 1234 da 0.012.

Come al solito si insiste molto sulla **comprensione del problema** (evidenza cartesiana). Lo studente deve convincersi che cercare di capire a fondo il problema dato è già risolverlo: la risoluzione dell'esercizio inzia proprio con la sua comprensione, dettagliata e profonda.

Alla luce ci ciò si vuole fornire una precisazione: nel prosieguo dell'esercizio si intenderà numero intero anche un qualsiasi reale avente la parte decimale uguale a zero. Ad esempio, il numero 2,000 (si è usata la virgola invece del punto decimale) è da intendersi, nell'ottica dell'esercizio, come numero intero o, se si preferisce, avente lo stesso valore dell'equivalente numero intero.

L'esercizio chiede che il problema venga risolto *aritmeticamente*, ovvero con le sole quattro operazioni aritmetiche. Non dobbiamo, cioè, lasciarci distrarre dall'organo della vista, che individua immediatamente il punto decimale o la virgola decimale e permette una risposta immediata. Lo studente deve immaginare di lavorare in coppia: un partner scrive su un foglio di carta il numero e, non potendolo vedere, si devono porre delle domande al partner per poter stabilire se il numero è intero o decimale. Non si può chiedere se c'è una virgola o un punto decimale, ma solo domande concernenti operazioni aritmetiche.

Non si tratta di un'operazione semplice, ma lo studente farebbe bene a cimentarsi in questo tipo di esercizio mentale al fine di affinare pian piano la propria capacità d'analisi. Si invita pertanto l'allievo a riflettere attentamente sul problema al fine di comprenderlo esaustivamente e a cercare di capire quali operazioni aritmetiche possono tornare utili nella risoluzione del problema.

Un modo piuttosto semplice per capire se un numero è intero o decimale potrebbe essere il seguente:

1. si supponga un numero r qualsiasi;
2. lo si moltiplichi per 10 e si chiami il risultato m;
3. si esegua la divisione intera per 10 di m e si chiami il risultato n (si esegua, cioè, $n = m/10$);
4. si sottragga n da r: se il risultato è 0, r è intero altrimenti è decimale.

[7]Come si possa essere sicuri che il numero sia effettivamente non negativo si vedrà nei prossimi esercizi.

Può essere che l'allievo abbia avuto difficoltà a pensare autonomamente un algoritmo simile a quello testè enunciato. Egli deve, però, convincersi che non sono gli strumenti che gli mancano, ma l'abitudine ad usarli!

Come procedere? Lo studente dovrebbe avere davanti agli occhi la faccia severa di Donald Ervin Knuth[8] e ricordarsi il suo ammonimento proposto a pag. 6: "Gli algoritmi vanno eseguiti, non studiati". Proviamo allora a immaginare due numeri, uno intero e uno decimale ed eseguire l'algoritmo proposto. Si supponga che il primo numero sia 123. Si avrà:

1. $r = 123$
2. $m = r \cdot 10 = 1230$
3. $n = 1230/10 = 123$ (divisione intera)
4. $r - n = 123 - 123 = 0 \Rightarrow r = intero$

Effettivamente sembra funzionare. Si propone un secondo tentativo con il numero 123.4:

1. $r = 123.4$
2. $m = r \cdot 10 = 1234$
3. $n = 1234/10 = 123$ (divisione intera)
4. $r - n = 123.4 - 123 = 0.4 \Rightarrow r = decimale$

I primi tre passi sono delle semplici elaborazioni: si può decidere se porle tutte in un'unica sequenza o mantenerle separate in tre sequenze distinte. Si opta per la seconda soluzione. La prima parte del diagramma di flusso è quindi simile alla seguente:

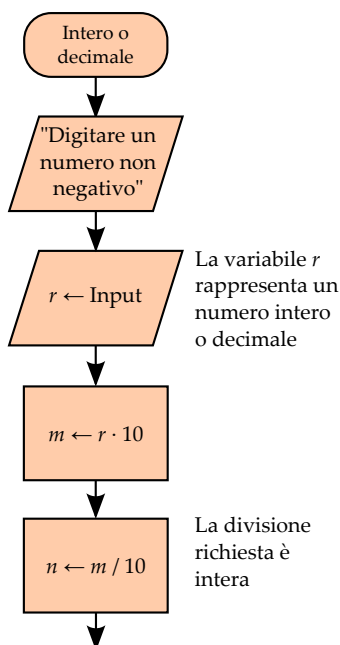



Figura 1.29: Intero decimale: Prime elaborazioni

[8]In realtà Knuth non ha per nulla la faccia severa, ma non fa niente. Si tratta di un'iperbole.

Il diagramma sembra l'esatta trasposizione grafica dell'algoritmo esposto in linguaggio naturale. L'unico appunto che si puà elevare consiste nel notare che il primo blocco è un blocco di Input e come tale è esplicitato mediante un parallelogramma. Le successive elaborazioni sono state esposte graficamente utilizzando le stesse variabili e le stesse notazioni usate nella descrizione dell'algoritmo fatta in linguaggio naturale.

Il passo successivo consiste nel trasporre graficamente il passo 4. Detto *step* è esposto in forma matematica come implicazione logica (*se ... allora*). Si tratta, quindi, di una selezione i cui rami di uscita devono condurre in altrettanti blocchi di Output ove di evidenziano i due possibili casi: intero o decimale. Il diagramma assume il seguente aspetto finale:

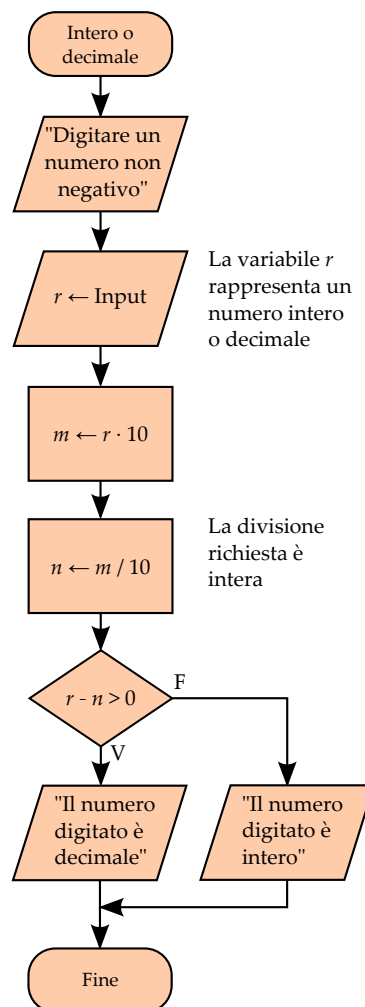



Figura 1.30: Intero decimale: Diagramma finale

Rimane da valutare l'esattezza del diagramma. Il miglior modo, dopo un accurato controllo teorico, consiste nell'eseguire l'algoritmo rappresentato dal

diagramma di flusso con 2-3 numeri interi e decimali. A mo' d'esercizio l'allievo potrebbe provare ad eseguire l'algoritmo con 12, 1.2 e con 0.

Si propongono i seguenti esercizi (simili a quello appena visto) non risolti.

1. ◊◊◊ **Valore assoluto.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero qualsiasi sulla tastiera del PC;
   (b) calcolare il valore assoluto del numero digitato;
   (c) stampare a video il valore assoluto del numero.
2. ◊◊◊ **Intero o decimale anche negativo.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero qualsiasi (anche negativo) sulla tastiera del PC;
   (b) valutare, utilizzando solo le quattro operazioni aritmetiche, se il numero è intero oppure decimale;
   (c) stampare a video se il numero è intero o decimale.
3. ◊◊◊ **Media aritmetica.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare due numeri naturali sulla tastiera del PC;
   (b) calcolare la media aritmetica dei due numeri;
   (c) stampare a video la media aritmetica.
4. ◊◊◊ **Maggiore fra due.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare due numeri relativi n ed m sulla tastiera del PC;
   (b) valutare quale dei ha il valor assoluto maggiore;
   (c) stampare a video il numero avente il maggior numero assoluto.
5. ◊◊◊ **Vicinanza allo zero.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare due numeri relativi n ed m sulla tastiera del PC;
   (b) valutare quale dei due sia più vicino allo zero;
   (c) stampare a video il numero più vicino allo zero.

Esercizio - ◊◊◊ Trova il massimo

Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:

1. digitare m numeri naturali sulla tastiera del PC (con $m > 0$);
2. valutare quale sia il maggiore degli m numeri digitati;
3. stampare a video il maggiore dei numeri digitati.

Soluzione

Si tratta di un classico dell'algoritmica per *beginners*. Sembra assolutamente innocente ma può presentare qualche minima difficoltà per chi è alle prime armi con gli algoritmi: dati m numeri dire qual è il maggiore.

Si suppongano i numeri 3, 21 e 7 (quindi con $m = 3$): cosa c'è di più semplice dello stabilire che 21 è il maggiore dei tre numeri?

Come al solito, il problema non è mai quello che appare ad una prima analisi. Non si tratta, infatti, di stabilire quale numero sia il maggiore, ma quale sia la procedura adottata per stabilirlo. Soprattutto pensando che si potrebbe avere $m = 1000$ o peggio.

Queste brevi riflessioni dovrebbero aver già gettato una certa luce sul problema (traduzione: il problema dovrebbe essere un po' più chiaro). La quantità dei numeri è data ma non nota e si deve trovare un algoritmo che funzioni sia per $m = 1$ sia per $m = 1000000$ (tanto per esagerare un po'). Il problema sembra effettivamente chiaro (*evidenza*). Meno chiaro è come impostare la risoluzione (*analisi*).

Se il problema richiede un livello di astrazione che non si è attualmente in grado di fornire, conviene fare degli esempi con numeri molto piccoli.

Esempio: come si risolve il problema se $m = 1$ oppure se $m = 2$? Anzi, conviene riformulare meglio la domanda: come risolvere il problema se $m = 1$ oppure se $m = 2$, *utilizzando in ambedue i casi la stessa procedura* (algoritmo)? Lo studente farebbe bene a dedicare, come al solito, un po' di tempo per rispondere a quest'ultima domanda.

Si potrebbe pensare in maniera *laterale*[9]: invece di pensare al risultato finale conviene pensare al risultato iniziale, ossia si potrebbe supporre che il risultato parziale, cioè il risultato definito *prima* di analizzare il valore dei vari numeri, sia 0, ovvero il più basso dei valori possibili (il testo parla di numeri naturali).

Si potrebbe pensare che il risultato sia una pallina con il numero dipinto sopra e di tenere in mano il risultato. Il risultato parziale deve ora essere confrontato, ad uno ad uno, con i (due) numeri. Si inizia con il primo: se il numero è maggiore di 0 (non può essere minore) si aggiorna il risultato parziale prendendo in mano la nuova pallina, altrimenti (potrebbero esserci più palline con lo 0) non si cambia la pallina.

A mo' d'esempio, si supponga che il primo numero sia 12. Si cambia pallina e si aggiorna il risultato parziale. Se $m = 1$ l'algoritmo termina per raggiunto limite dei confonti operati (ossia 1). Se $m = 2$ si procede con un secondo confronto identico a quello precedente: se il nuovo numero è maggiore di quello già acquisito lo si cambia, altrimenti si tiene il vecchio numero. Tale metodo fuziona per qualsiasi valore di m, supponendo anche che i numeri estratti possano essere doppi, tripli (ad esempio, due 12 o tre 31), ecc.

Visto che l'idea sembra reggere, si propone un esempio un po' più lungo. Si suppongano i seguenti numeri di cui si deve trovare il massimo: 5, 2, 5, 14, 8, 0, 14, 17, 3. Supponendo di indicare il singolo numero con n ed il risultato parziale con ris e supponendo anche che inizialmente si abbia $ris = 0$, si possono inziare i confronti.

Il primo confronto è fra 0, ossia l'attuale risultato parziale, e 5. 5 è maggiore di 0, quindi il risultato parziale va aggiornato con 5. Il secondo confronto è fra 5 e 2. In questo caso il risultato parziale non va aggiornato perché il nuovo numero è minore.

Anche il terzo confronto non prevede alcun aggiornamento del risultato parziale, dato che il nuovo numero è un "doppione" ed è uguale a *ris*.

[9] Il *pensiero laterale* è una modalità di risoluzione dei problemi logici coniato dallo psicologo Edward De Bono.

Continuando con lo stesso criterio per tutti i numeri, è possibile redarre la seguente tabella:

n	Confronto	Esito	ris
5	$5 > 0$	Si	5
2	$2 > 5$	No	5
5	$5 > 5$	No	5
14	$14 > 5$	Si	14
8	$8 > 14$	No	14
0	$0 > 14$	No	14
17	$17 > 14$	Si	17
3	$3 > 17$	No	17

Tabella 1.4: Tabella di ricerca del massimo

Per cui il risultato finale è 17.

Lo studente dovrebbe ora riflettere sulla soluzione proposta per valutare se è stata capita oppure no. In caso affermativo si può procedere nella *sintesi* con la stesura del diagramma di flusso. L'inizializzazione dell'algoritmo potrebbe essere simile alla seguente:

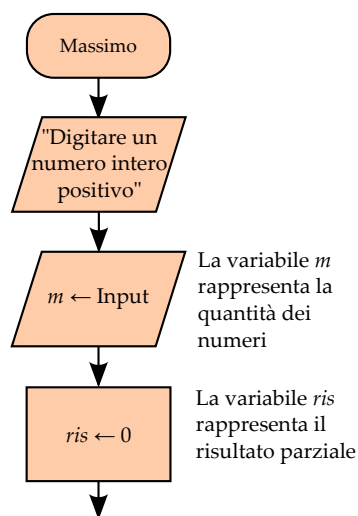



Figura 1.31: Massimo: Inizializzazione

Questa prima parte non dovrebbe creare problemi di nessun tipo.

Lo studente dovrebbe ora frenare la voglia di continuare subito la stesura del diagramma e, data l'ovvia presenza di un'iterazione (sono stati fatti molti confronti, tutti uguali), porsi una domanda: "Di che iterazione si tratta?"

Lo studente dovrebbe porsi tale domanda ogni volta che sta per tracciare un'iterazione. Normalmente questa riflessione viene serenamente ignorata, con tanti saluti all'atteggiamento critico, ai "perché" dell'insegnante, al "saper essere" e a Cartesio. Si tratta, invece, di una domanda assolutamente fondamentale dal punto di vista metodologico, altrimenti prevale l'intuizione e la

ragione viene messa a dormire[10]. Siccome il numero di cicli dell'iterazione è noto ed è uguale ad m[11], si utilizzerà un'iterazione con numero di cicli definito a priori[12], da cui si evince il diagramma di fig. 1.32.

Figura 1.32: Massimo: Diagramma finale

[10]Il sonno della ragione genera diagrammi mostruosi.

[11]Anche in questo caso il controllo $m > 0$ verrà affrontato nel prossimo esercizio guidato.

[12]Un utile esercizio consiste nell'individuare l'iterazione nel diagramma di fig. 1.32, nonché le parti che la compongono (selezione, sequenza e decremento dell'indice). A tal proposito si noti che il *layout* grafico è piuttosto infelice, ma permette una facile identificazione delle strutture.

Ora, non rimane altro che la verifica del diagramma (*enumerazione*). Lo si può fare con gli stessi numeri già usati per validare l'algoritmo. E' un esercizio che si lascia all'allievo.

Di seguito si propongono i seguenti esercizi (simili a quello appena visto) non risolti.

1. ◊◊◊ **Trova il massimo relativo.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare m numeri relativi sulla tastiera del PC (con $m > 0$);
   (b) valutare quale sia il maggiore degli m numeri digitati;
   (c) stampare a video il maggiore dei numeri digitati.
2. ◊◊◊ **Media aritmetica.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare m numeri relativi sulla tastiera del PC (con $m > 0$);
   (b) calcolare la media aritmetica degli m numeri digitati;
   (c) stampare a video la media aritmetica dei numeri digitati.
3. ◊◊◊ **Trova il numero.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare m numeri relativi sulla tastiera del PC (con $m > 0$);
   (b) valutare se fra gli m numeri vi sia il numero intero m;
   (c) stampare a video la presenza/assenza di tale numero.
4. ◊◊◊ **Calcola l'intervallo.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare m numeri relativi sulla tastiera del PC (con $m > 0$);
   (b) calcolare la differenza fra il massimo valore digitato ed il minimo valore digitato;
   (c) stampare a video la differenza calcolata.
5. ◊◊◊ **Ricorrenza.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare m numeri relativi sulla tastiera del PC (con $m > 0$);
   (b) calcolare quante volte compare, fra gli m numeri digitati, il numero intero m;
   (c) stampare a video il numero delle ricorrenze del numero m.

Esercizio - ◊◊◊ Digita il numero

Il presente esercizio illustra come si possa controllare che un numero digitato da tastiera abbia effettivamente i requisiti richiesti dal testo del problema.

Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:

1. chiedere all'utente di digitare un numero intero $m > 0$;
2. digitare il numero m;
3. verificare che il numero abbia i requisiti richiesti;
4. tornare al punto 1 se i requisiti non sono soddisfatti.

Soluzione

Se m non dovesse essere maggiore di 0, l'utente deve essere invitato a ridigitare il numero finchè non si abbia $m > 0$. Anche questo problema è molto frequente e l'allievo avrà molte occasioni di incontrarlo. E' quindi bene che comprenda a fondo come il problema possa essere risolto.

Le specifiche fornite dall'esercizio sembrano piuttosto chiare e anche in seguito ad una riflessione non superficiale non paiono presenti punti di ambiguità. Si può passare quindi subito alla fase di analisi.

La "suddivisione" del problema in sottoproblemi, in pratica, è già data dal testo stesso: si possono immaginare 4 parti distinte:

1. una richiesta;
2. una immissione da tastiera;
3. una verifica;
4. un eventuale ritorno al punto 1 se i requisiti forniti dalla richiesta non sono soddisfatti. In caso contrario l'algoritmo termina.

E' abbastanza palese che l'algoritmo ha la struttura di un'iterazione, quindi bisogna valutare quale tipo di iterazione debba essere usata. Si invita pertanto l'allievo ad una non superficiale riflessione sull'argomento.

Il *focus* del problema è il corpo del ciclo. Prima di parlare di "test finale" o "test iniziale" è bene concentrarsi su quali azioni debbano essere ripetute.

Si ritiene non ci siano eccessivi dubbi sul fatto che le azioni da ripetere siano le prime due (la richiesta di immissione del numero e la sua digitazione). Quindi, sostanzialmente, ci si deve chiedere:

- "So quante volte dovrò entrare nel corpo del ciclo?"
- "E' possibile che non si entri mai nel corpo del ciclo?"
- "Devo entrare nel corpo del ciclo almeno una volta?"

Al primo "Sì" ci si deve fermare e si noti che l'ordine delle domande non è casuale.

Si ritiene di sapere quante volte si entrerà nel corpo del ciclo? Se così fosse si dovrebbe utilizzare un'iterazione a numero di cilci definito a priori. Non sembra, però, che si possa rispondere affermativamente a tale domanda. Non si può sapere quante volte l'utente "sbaglierà" di immettere il valore di m. L'utente potrebbe voler testare l'algoritmo validandolo con varie immissioni errate. L'utente potrebbe "divertirsi" ad immettere numeri che non soddisfano i requisiti richiesti. L'utente potrebbe essere semplicemente distratto, oppure potrebbe immettere correttamente il numero al primo tentativo. Tutto ciò, però, non ci è noto a priori.

Si ritiene che sia possibile non entrare mai nel corpo del ciclo? In tal caso l'azione di immissione del numero non verrebbe mai espletata e non si sarebbe in possesso di alcun numero, nè soddisfacente i requisiti richiesti nè sprovvisto dei medesimi.

Ciò significa che *si deve entrare almeno una volta nel corpo del ciclo* e che non si sa quate volte lo si farà (ossia quanti errori di immissione si commetteranno). L'affermazione vera è quindi la terza e il tipo di iterazione è quella a test finale con numero di cicli non definito a priori.

Alla luce di quanto fin qui detto, un diagramma di flusso assolutamente *standard* potrebbe quindi essere il seguente:

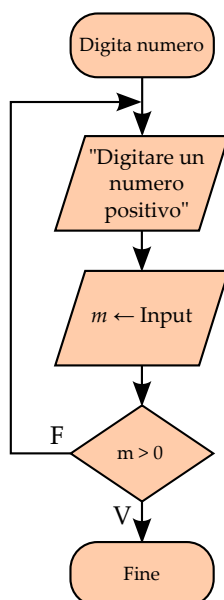



Figura 1.33: Digita numero: Diagramma finale

In questo caso la sequenza e la modifica del predicato praticamente coincidono con l'Input del dato. Se i requisiti richiesti ed enunciati dal blocco di Output sono soddisfatti si esce dall'algoritmo e il risultato è nella variabile m, altrimenti si rientra nel corpo del ciclo, la richiesta viene nuovamente formulata e l'Input viene nuovamente effettuato.

In caso di errore si potrebbe riformulare in altro modo la frase di cortesia. Ciò significa che la frase attuale dovrebbe essere posta prima del ciclo e la frase la visualizzare in caso di errore sul ramo di ritorno del ciclo.

Questo tipo di diagramma è praticamente "pronto per l'uso" in moltissime situazioni diverse: è sufficiente modificare la richiesta ed il blocco di selezione, modificando il predicato.

E' piuttosto facile verificare che se $m > 0$ si esce subito dall'algoritmo e che il corpo del ciclo viene ripetuto finché il predicato non è soddisfatto. L'allievo può anche verificare che dal punto di vista logico le altre due iterazioni sono soddisfacenti della presente.

Di seguito si propongono i seguenti esercizi (simili a quello appena visto) non risolti.

1. ◊◊◊ **Numero dispari.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) chiedere all'utente di digitare un numero intero dispari;
   (b) digitare il numero m;
   (c) verificare che il numero abbia i requisiti richiesti;
   (d) tornare al punto 1 se i requisiti non sono soddisfatti.

2. ◊◊◊ **Numero divisibile per 3.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) chiedere all'utente di digitare un numero intero divisibile per 3;
   (b) digitare il numero m;
   (c) verificare che il numero abbia i requisiti richiesti;
   (d) tornare al punto 1 se i requisiti non sono soddisfatti.
3. ◊◊◊ **Numero compreso fra 3 e 7.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) chiedere all'utente di digitare un numero intero compreso fra 3 e 7;
   (b) digitare il numero m;
   (c) verificare che il numero abbia i requisiti richiesti;
   (d) tornare al punto 1 se i requisiti non sono soddisfatti.
4. ◊◊◊ **Numero di 3 cifre.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) chiedere all'utente di digitare un numero intero di 3 cifre;
   (b) digitare il numero m;
   (c) verificare che il numero abbia i requisiti richiesti;
   (d) tornare al punto 1 se i requisiti non sono soddisfatti.
5. ◊◊◊ **Somma e differenza.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) chiedere all'utente di digitare un numero intero di 2 cifre tale che:
      i. la somma delle singole cifre sia pari a 8;
      ii. la differenza delle singole cifre sia pari a 2.
   (b) digitare il numero m;
   (c) verificare che il numero abbia i requisiti richiesti;
   (d) tornare al punto 1 se i requisiti non sono soddisfatti.

Esercizio - ◊◊♦ Somma cifre

Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:

1. chiedere all'utente di digitare un numero decimale d qualsiasi;
2. digitare il numero d;
3. stampare a video la somma delle cifre della parte intera e della parte decimale di m.

Soluzione

Questo è un problema con una stella, il che significa che si è aumentato leggermente la difficoltà dell'esercizio. Quindi si dovrà raddoppiare l'attenzione e seguire scrupolosamente un metodo. In queste pagine si insiste con particolare enfasi sul metodo cartesiano (spiegato diffusamente in altra documentazione: se lo studente ne è privo potrà richiederla via mail o direttamente all'insegnante), ma si ricorda che Cartesio stesso esorta a costruirsi un proprio metodo di ricerca della soluzione.

Le richieste dell'esercizio potrebbero non essere chiarissime. Si chiede di immettere da tastiera un numero decimale qualsiasi e di sommare le cifre della parte intera e le cifre della parte decimale. Se, ad esempio, il numero digitato è 123.45, la somma delle cifre della vale $1 + 2 + 3 + 4 + 5 = 15$.

Ora che il problema sembra chiaro ci si può concentrare sull'analisi. Sarà sicuramente la parte più difficile dell'esercizio, per almeno due motivi: 1) l'esercizio ha una stella, quindi è un po' più difficile dei precedenti; 2) nel testo non è proposto alcun algoritmo di risoluzione.

Questo secondo motivo dovrebbe attirare l'attenzione dello studente. Frequentemente si sente dire in aula: "Ho capito il problema, ma non so come risolverlo". Riuscire a fornire degli strumenti all'allievo per la risoluzione dei problemi potrebbe essere un grosso passo in avanti nell'*imparare ad imparare*. Il presente esercizio si propone di aiutare lo studente in tal senso.

Lo studente potrebbe/dovrebbe fare una considerazione piuttosto ovvia: "Se il professore mi ha dato questo problema, significa che lo posso affrontare/risolvere".

Purtroppo, invece, lo studente pensa spesso: "Se avessi studiato avrei gli strumenti per affrontare l'esercizio, ma siccome non ho studiato (abbastanza) questi strumenti non li possiedo".

Cerchiamo, innanzi tutto, di rincuorare lo studente: premesso che studiare è il primo passo da compiere per acquisire la conoscenza, quindi va considerato come condizione necessaria non sufficiente per poter affrontare la moltitudine di problemi che lo studente incontrerà a scuola e nella vita, la quantità (enorme) di conoscenze accumulate nel corso della propria vita scolastica dovrà ben servire a qualcosa. Non è mica possibile che i problemi presentati si debbano risolvere sempre con l'argomento dell'ultima lezione.

Il presente esercizio, ad esempio, è perfettamente risolvibile con le conoscenze acquisite nella scuola primaria. La difficoltà, come spesso accade, non sta nella conoscenza, ma nell'organizzazione della stessa. Umberto Eco ha avuto modo di dire (più o meno): "Cultura significa sapere le cose adesso, quando serve, non fra 10 minuti". Inoltre, è molto più importante saper affrontare situazioni nuove, piuttosto che "risolvere" ottusamente esercizi triti e ritriti.

Non si vuole proporre qui una trattazione delle tecniche di risoluzione dei problemi. Una breve introduzione al *problem solving* è data negli appunti "Problem Solving Strategies" dello stesso autore delle presenti pagine.

Insomma, lo studente farebbe bene ad avere un atteggiamento positivo e non remissivo: quasi sempre possiede già le conoscenze. Si tratta di "condurle fuori" dalla propria mente (*educere*) e dar loro forma, ma ci sono già.

Una prima azione, di carattere generale, che l'allievo può fare è "elencare" tutte le conoscenze in suo possesso che potrebbero essere utili nella soluzione del problema, un'azione simile, quindi, al *brain storming*. Si tratta di "estrarre" le singole cifre di un numero, quindi si potrebbero essere utili ...

- le quattro operazioni aritmetiche;
- i sistemi di equazioni;
- la rappresentazione polinomiale dei numeri reali;
- ..., ecc.

Sicuramente le quattro operazioni aritmetiche potrebbero tornare utili, dato che si sta parlando di numeri.

Per dovere di completezza, si sono elencati anche i sistemi di equazioni, perché rappresentano uno strumento potentissimo nella rtisoluzione dei problemi numerici, anche se, francamente, c'è difficoltà ad individuare il numero delle variabili e a trovare tante equazioni quante sono le variabili. Troppo complesso.

La rappresentazione in forma polinomiale di un numero offre degli spunti di riflessione. Ogni cifra è un termine del polinomio, espressa come valore e peso. Ad esempio, il numero 12.34_{10} può essere rappresentato in forma polinomiale come

$$12.34 = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2} \tag{1.1}$$
$$12.34 = 10 + 2 + 0.3 + 0.04 \tag{1.2}$$

Tutto ciò, però, non ci aiuta ad isolare le singole cifre, anche se l'impressione è che qualche piccolo passo in avanti è stato fatto. Se si incontrano difficoltà nell'astrarre un problema complesso, conviene fare degli esempi concreti su numeri molto piccoli, risolvendo il problema per gradi.

Si potrebbe ipotizzare un numero molto semplice: per iniziare si potrebbe supporre un numero intero di sole due cifre, ad esempio, 12.

Se si volesse conoscere la cifra meno significativa del numero, sarebbe sufficiente dividere, mediante divisione intera, il numero per 10: il resto della divisione corrisponderebbe alla cifra meno significativa:

$$12 : 10 = 1 \text{ (quoziente) con resto } 2 \tag{1.3}$$

Ripetendo l'operazione 1.3 sul quoziente si ottiene:

$$1 : 10 = 0 \text{ (quoziente) con resto } 1 \tag{1.4}$$

Se si ripete l'operazione 1.3 finché il quoziente non diventa uguale a 0, i singoli resti delle divisioni rappresentano le cifre del numero.

Si può anche specificare meglio come individuare il resto. Volendo "estrarre" la cifra meno significativa di $m = 12$ usando una calcolatrice fornita delle sole quattro operazioni aritmetiche, si potrebbe operare nel seguente modo:

$$m \leftarrow 12 \tag{1.5}$$
$$n \leftarrow m : 10 \quad \text{(divisione intera)} \tag{1.6}$$
$$n \leftarrow n \cdot 10 \tag{1.7}$$
$$c \leftarrow m - n \tag{1.8}$$

Se il numero fosse intero, l'esercizio sarebbe quindi risolto. Il problema diventa ora: "come si può fare diventare intero un reale?" o meglio come si può "portare" nella parte intera le cifre decimali. Naturalmente moltiplicando ripetutamente il numero per 10.

Quando ci si deve fermare? Quando la parte decimale è formata da soli zeri. Come riconoscere questa situazione? Si chiede allo studente di riflettere su un possibile algoritmo.

◊◊◊

Il problema è già stato affrontato nell'esercizio **Intero Decimale**, in cui si chiedeva di identificare se un numero qualsiasi aveva la parte decimale posta a zero o meno. Si supponga, ad esempio, il numero $d = 12.34$. Si potrebbe:

1. moltiplicare il numero per 10, tale $m = d \cdot 10$;
2. dividere (divisione intera) m per 10, tale che $n = m : 10$;
3. se il predicato $d - n > 0$ è vero, significa che la parte decimale di d è maggiore di zero. In tal caso assegnare a d il valore di m e tornare al punto 1. Se invece il predicato è falso, la parte decimale di d è uguale a zero e l'algoritmo termina.

Le predette operazioni sono tutte elementari e assolutamente facenti parte del bagaglio di conoscenze di qualsiasi studente delle medie superiori (o delle medie inferiori). Ciò nonstante è possibile che molti studenti abbiano trovato difficoltà, anche grosse, ad *utilizzare* proficuamente queste conoscenze. Non si devono scoraggiare, ma insistere: il lavoro porta sempre frutti, anche se volte non sono frutti immediati.

E' corretto verificare se l'algoritmo funziona. Si supponga a tal fine il numero 12.34 e si applichi l'algoritmo a detto numero:

$$d \leftarrow 12.34 \quad (1.9)$$
$$m \leftarrow d \cdot 10 \quad (1.10)$$
$$n \leftarrow m : 10 \quad (1.11)$$
$$d - n > 0 \quad \rightarrow \quad 12.34 - 12 > 0 \text{ (predicato vero: continua)} \quad (1.12)$$
$$d \leftarrow m \quad (1.13)$$
$$m \leftarrow d \cdot 10 \quad (1.14)$$
$$n \leftarrow m : 10 \quad (1.15)$$
$$d - n > 0 \quad \rightarrow \quad 123.4 - 123 > 0 \text{ (predicato vero: continua)} \quad (1.16)$$
$$d \leftarrow m \quad (1.17)$$
$$m \leftarrow d \cdot 10 \quad (1.18)$$
$$n \leftarrow m : 10 \quad (1.19)$$
$$d - n > 0 \quad \rightarrow \quad 1234 - 1234 > 0 \text{ (predicato falso: fine)} \quad (1.20)$$

Dopo la 1.20 il numero ha la parte decimale uguale a zero e può, sotto certi aspetti, essere visto come un numero intero. Inoltre, è sufficiente contare il numero delle volte in cui il predicato era vero per sapere di quante cifre decimali era composto il numero originale.

In tal modo l'analisi è terminata. Si è riusciti ad isolare le singole cifre di un numero intero (non le abbiamo sommate, ma avremmo potuto farlo. Si tratta di un dettaglio), e siamo riusciti a ricondurre un numero decimale ad un numero intero. Applicando questi due *step* è possibile sommare tutte le cifre di un qualsiasi numero razionale non periodico, compresi, ad esempio, i numeri 0.0 e 123456789.987654321.

In caso di dubbi l'allievo farebbe bene a rileggere la parte poco chiara e a *eseguire*, con carta e penna, gli algoritmi presentati, magari riproducendo prima gli esempi forniti e poi proponendo esempi propri. Eseguendo gli algoritmi

correttamente, i dubbi dovrebbero pian piano venir meno e agevolare una completa comprensione delle procedure usate. Se gli algoritmi vengono applicati in modo che i risultati non siano corretti, l'allievo deve ritornare sui suoi passi e ristudiare da capo l'esercizio.

Se tutto è chiarto, è possibile affrontare la sintesi, ovvero la realizzazione del diagramma di flusso. Si traccerà dapprima un diagramma molto semplice e si dettaglieranno in un secondo tempo le singole sequenze che necessitano approfondimenti.

Un possibile diagramma di flusso che sia frutto dei ragionamenti fin qui elaborati potrebbe essere il seguente:

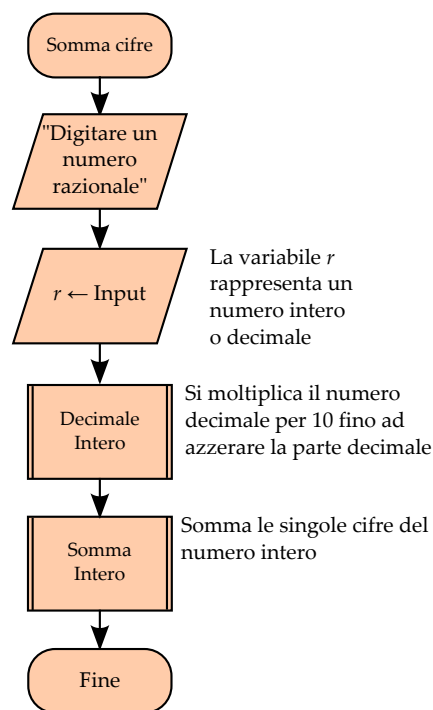



Figura 1.34: Somma cifre: Primo diagramma

I primi due blocchi sono di Output e di Input. Il primo è la frase di cortesia che chiede all'utente l'insertimento del numero, mentre il secondo è il blocco di input del numero razionale.

I due blocchi che seguono sono due *subroutines* che implementano l'algoritmo vero e proprio: un primo blocco che trasforma il numero razionale in numero intero[13] ed un secondo blocco che effettua la somma delle varie cifre del numero intero.

Nel caso in cui in numero digitato abbia già la parte decimale posta a zero, il blocco di elaborazione **Decimale Intero** viene comunque eseguito, limitandosi a constatare che il numero è già intero o, se si preferisce, con parte decimale uguale a zero.

[13]In realtà viene solo azzerata la parte decimale del numero razionale.

Il blocco **Decimale Intero** può essere dettagliato meglio, come esposto durante l'analisi. Dall'esecuzione dell'algoritmo si è appreso della presenza di un'iterazione. Dobbiamo, quindi, chiederci di che tipo di iterazione si tratti: se a numero di ciclo definito; se a numero di cicli non definito e a test iniziale, oppure a test finale.

Come nell'esercizio precedente dobbiamo porci, nell'ordine, le solite tre domande:

- "So quante volte dovrò entrare nel corpo del ciclo?"
- "E' possibile che non si entri mai nel corpo del ciclo?"
- "Devo entrare nel corpo del ciclo almeno una volta?"

Alla prima domanda si dovrà rispondere "No", dato che non si conosce, prima di entrare nel ciclo, il numero delle cifre decimali.

Anche alla seconda domanda si dovrà rispondere "No", dato che almeno una volta si dovrà entrare nel corpo del ciclo per valutare se il numero ha cifre decimali oppure no.

Si userà, quindi, un'iterazione a test finale. Un esempio potrebbe essere il seguente:

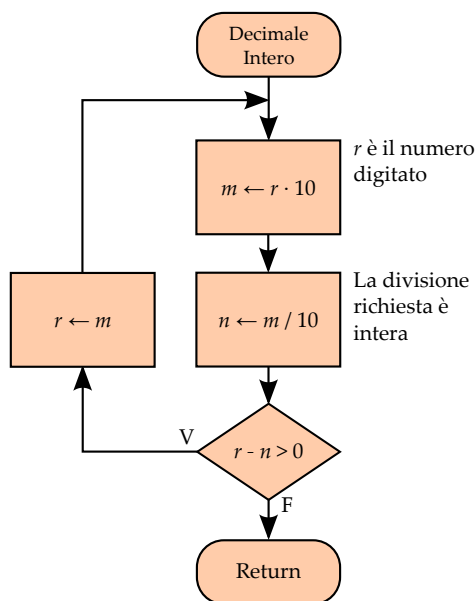



Figura 1.35: Decimale Intero: Diagramma finale

Il diagramma di fig. 1.35 implementa alla lettera i passaggi da 1.10 a 1.20 a pagina 47 già ampiamente illustrati, per cui non si ritiene necessario commentare ulteriormente il diagramma.

L'unica nota che si ritiene utile riguarda non tanto l'algoritmo quanto l'uso della *subroutine*. Come già detto nella sezione 1.8 essa serve a non sovraccaricare eccessivamente il diagramma e a riutilizzare quelle sequenze che vengono riutilizzate in più punti del diagramma.

Nel presente diagramma essa è stata utilizzata per dettagliare per gradi l'algoritmo e renderne più fluida la lettura.

La seconda *subroutine* è la **Somma Intero**, il cui algoritmo è stato spiegato in occasione dei passaggi da 1.6 a 1.8 a pagina 46. Anche in questo caso non si ritiene necessario aggiungere commenti. Il diagramma della *subroutine* è il seguente:

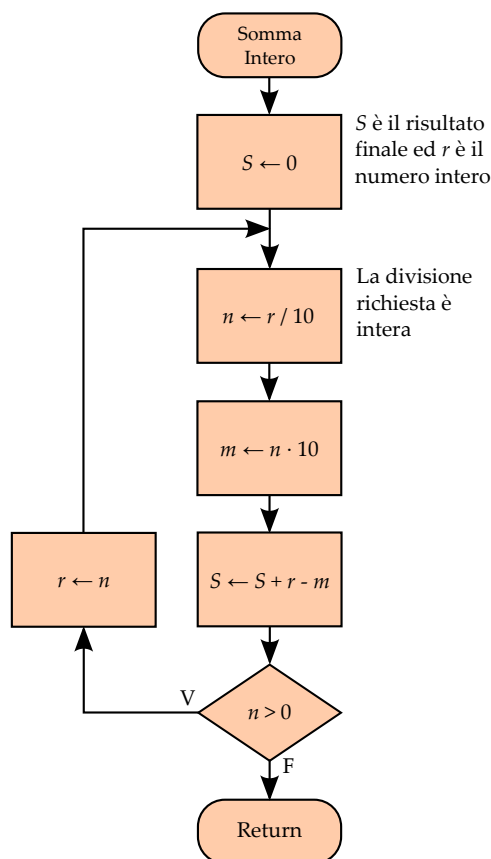



Figura 1.36: Somma Intero: Diagramma finale

Rimane da effettuare l'enumerazione cartesiana, ovvero il controllo finale dell'elaborato.

Tale azione può essere efficacemente svolta eseguendo l'algoritmo, così com'è descritto con il diagramma di flusso, con 2-3 numeri significativi. Si lascia la verifica allo studente a mo' d'esercizio. L'allievo dovrebbe dichiarare il criterio di scelta dei numeri usati per il collaudo del diagramma.

Di seguito si propongono i seguenti esercizi non risolti.

1. ◊◊◊ **Cifra massima.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) individuare la maggiore delle cifre del numero;
   (c) stampare a video la cifra massima.

2. ♢♢♢ **Numero di zeri.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) calcolare il numero di zeri contenuti nel numero;
   (c) stampare a video il numero di zeri calcolato.
3. ♢♢♢ **Numero cifre.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) calcolare il numero di cifre del numero digitato;
   (c) stampare a video il numero di cifre calcolato.
4. ♢♢♢ **Numero ruotato.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) "ruotare" il numero in modo che la cifra meno significativa diventi la più significativa e viceversa. Da 1234, si deve ottenere 4321;
   (c) stampare a video il numero "ruotato".
5. ♢♢♦ **Numero palindromo.** Si chiede la stesura del diagramma di flusso che descrive le seguenti azioni:
   (a) digitare un numero intero sulla tastiera del PC;
   (b) verificare se il numero è palindromo o meno. Sono palindromi quei numeri che sono uguali sia leggendoli sia da destra che da sinistra, come ad esempio 1234321. Per risolvere l'esercizio si consiglia di ridurre a *subroutine* i diagrammi dei due esercizi precedenti;
   (c) stampare a video se il numero è palindromo o meno.

Esercizio - ♢♢♦ Fibonacci

Si chiede la stesura del diagramma di flusso che, dato $n \in \mathbb{N}$, stampi a video l'ennesimo valore della successione di Fibonacci[14]. Essa è espressa come:

$$F_n = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F_{n-1} + F_{n-2}, & \text{se } n > 1. \end{cases} \tag{1.21}$$

Soluzione

Apri la mente a quel ch'io ti paleso
e fermalvi entro; ché non fa scïenza,
sanza lo ritenere, avere inteso.

Dante, Paradiso - Canto V

[14]Leonardo Pisano, detto Fibonacci (*filius Bonacci*) fu un importante matematico toscano che visse fra il XII e il XIII secolo. Teorizzo la sucessione che prende il suo nome nel trattato *Liber abaci* del 1202.

Non basta Cartesio: anche Dante deve dire la sua e va addirittura oltre. Il Sommo Poeta ammonisce[15] che non basta capire, ma bisogna anche trattenere la conoscenza.

Quindi anche questa volta ci si deve soffermare sul testo e cercare di capirlo a fondo prima di analizzare il problema.

Siccome l'argomento potrebbe mettere in imbarazzo lo studente, si forniscono ulteriori chiarimenti circa le successioni. Tecnicamente si definisce la **successione numerica** come una *funzione a valori reali avente per dominio l'insieme dei numeri naturali* $\mathbb{N}$.

A differenza degli insiemi, nelle successioni è importante l'ordine dei singoli *elementi*, la posizione è normalmente indicata da un *pedice*. Quindi F_0 è il primo elemento (e vale 0), F_1 il secondo (e vale 1) e F_n l'ennesimo elemento della successione.

Un qualsiasi elemento F_n (con $n > 1$) della funzione 9.1 è sempre calcolabile mediante la somma dei due elementi precedenti ($F_n = F_{n-1} + F_{n-2}$), per cui, ad esempio, i primi 10 elementi della successione sono:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad ...$$

Chiariamo ulteriormente il procedimento per calcolare gli elementi della successione di Fibonacci. I primi due elementi della successione sono dati dalla 9.1:

$$F_0 = 0 \tag{1.22}$$

$$F_1 = 1 \tag{1.23}$$

Il terzo elemento, ossia F_2, è dato dalla somma dei precedenti due, ovvero F_1 e F_0, quindi:

$$F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{1.24}$$

Il quarto elemento, ossia F_3, è dato dalla somma dei precedenti due, ovvero F_2 e F_1, quindi:

$$F_3 = F_2 + F_2 = 1 + 1 = 2 \tag{1.25}$$

Il quinto elemento, F_4, è dato dalla somma di F_3 e F_2, quindi:

$$F_4 = F_3 + F_2 = 2 + 1 = 3 \tag{1.26}$$

Il sesto elemento, F_5, è dato dalla somma di F_4 e F_3, quindi:

$$F_5 = F_4 + F_3 = 3 + 2 = 5 \tag{1.27}$$

e così via.

Quindi l'ennesimo elemento della successione è dato dal calcolo della seguente espressione:

$$F_n = F_{n-1} + F_{n-2} \tag{1.28}$$

a patto che $n > 1$.

Dovrebbe quindi essere chiaro come si forma la successione. Se si volesse conoscere il valore del 19_{esimo} elemento della successione, bisognerebbe continuare il procedimento fino all'elemento F_{18}. E' abbastanza evidente che per

[15] In realtà a "parlare" non è Dante, ma Beatrice cha ammonisce il poeta.

conoscere il valore numerico dell'ennesimo elemento della successione si deve procedere per somme successive iniziando sempre dai primi due valori, ossia 0 e 1. L'espressione 9.1 dovrebbe ora essere piuttosto chiara.

Anche dal punto di vista analitico non dovrebbero esserci eccessivi dubbi. Si tratta di inizializzare due valori (F_0 e F_1) e poi sommare ripetutamente i due valori precedenti fino all'elemento voluto.

Ciò significa eseguire un ciclo. Siccome il numero delle somme è calcolabile si utilizzerà un'iterazione con numero di cicli definito a priori. Supponendo di voler calcolare il valore di F_n, il calcolo del numero di cicli è dato dalla relazione

$$nc = n + 1 - 2 = n - 1 \tag{1.29}$$

Se, ad esempio, si volesse calcolare il valore di F_{18}, il numero di somme sarebbe dato da

$$nc = n - 1 = 18 - 1 = 17 \tag{1.30}$$

E' quindi possibile passare alla sintesi e tracciare un primo diagramma di flusso dell'algoritmo di calcolo dell'ennesimo elemento della successione di Fibonacci.

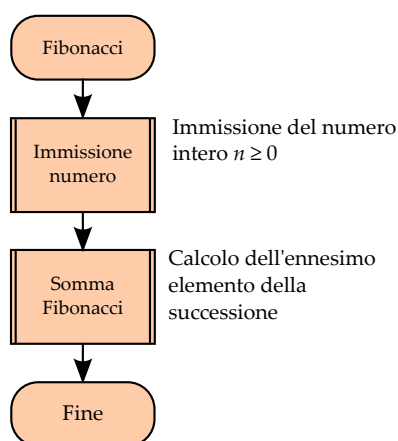



Figura 1.37: Fibonacci: Primo diagramma

Il primo blocco del diagramma di flusso è dato dall'immissione del numero. Si è deciso di utilizzare una *subroutine*, dato che la sequenza è relativamente complessa: si tratta sicuramente di un'iterazione dato che potrebbero esserci degli errori durante l'immissione del numero (potrebbe essere immesso, ad esempio, un numero negativo). Non è quindi una cattiva scelta quella di tracciare un primo diagramma grossolano, poco dettagliato e fornire i dettagli in altri diagrammi.

Analogo discorso vale per il blocco **Somma Fibonacci**. Esso rappresenta il blocco di elaborazione del vero e proprio algoritmo di Fibonacci, in cui viene graficamente rappresentata l'elaborazione della successione. In Input il blocco ha il numero fornito in Output dal blocco **Immissione numero** e fornisce, a sua volta l'ennesimo valore della successione di Fibonacci.

Il diagramma di fig. 1.37 può e deve essere, quindi, ulteriormente dettagliato. Si propone per primo il diagramma di flusso relativo all'immissiione del numero non negativo, come evidenziato in fig. 1.38.

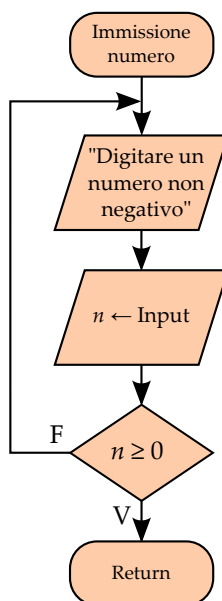



Figura 1.38: Fibonacci: Immissione numero

Il diagramma di flusso è del tutto simile a quello già illustrato in fig. 1.33 a pagina 43. Ciò basti a far capire allo studente come gli esercizi proposti rappresentino situazioni tipiche che facilmente possono essere riutilizzate. Non si ritiene, quindi, di dover commentare ulteriormente il diagramma.

I commenti sono, invece, doverosi per il diagramma di fig. 1.39 a fronte che implementa l'algoritmo vero e proprio della successione di Fibonacci. La prima cosa che colpisce è la scelta multipla. Essa è formata dalle prime due selezioni, quelle i cui predicati sono rispettivamente $n = 0$ e $n = 1$. Come già detto in precedenti esercizi, se si deve operare una scelta multipla di m distinti casi, sono sufficienti $m - 1$ selezioni.

Nel presente caso, sono quindi sufficienti 2 selezioni per produrre le scelte ai casi 1, 2 e 3 indicati nel diagramma di flusso. Nei casi 1 e 2 vengono semplicemente prodotti i risultati $c = 0$ e $c = 1$.

Più interessante è il caso 3, ove si ha $n > 1$ e viene applicata l'espressione $F_n = F_{n-1} + F_{n-2}$. Il caso 3 evidenzia, come prima azione, l'inizializzazione delle variabili a e b. Esse rappresentano i due casi iniziali: $a = F_0$ e $b = F_1$.

Si noti, poi, che l'iterazione usata è a numero di cicli definito a priori, anche se non appare subito evidente, *mancando il caricamento dell'indice*. In realtà tale caricamento non è necessario, dato che viene usato il numero n come indice di ciclo. Questa pratica è piuttosto diffusa e non deve stupire.

Dopo l'inizializzazione viene effettuato il test (infatti si tratta di un'iterazione a test iniziale). La prima volta il predicato è sicuramente vero, dato che il

caso 3 prevede proprio che $n > 1$. Viene quindi effetttuata la somma $c \leftarrow b + a$ ossia $F_n \leftarrow F_{n-1} + F_{n-2}$.

Il prossimo blocco è quello concettualmente più importante: vengono aggiornate le variabili a e b (altrimenti si farebbe sempre la stessa somma). Si noti che l'ordine di aggiornamento è importante: *prima* si aggiorna la variabile b e *dopo* la variabile a. Il decremento dell'indice può essere effettuato in qualsiasi momento. Il ciclo si interrompe quando n diventa uguale a 1.

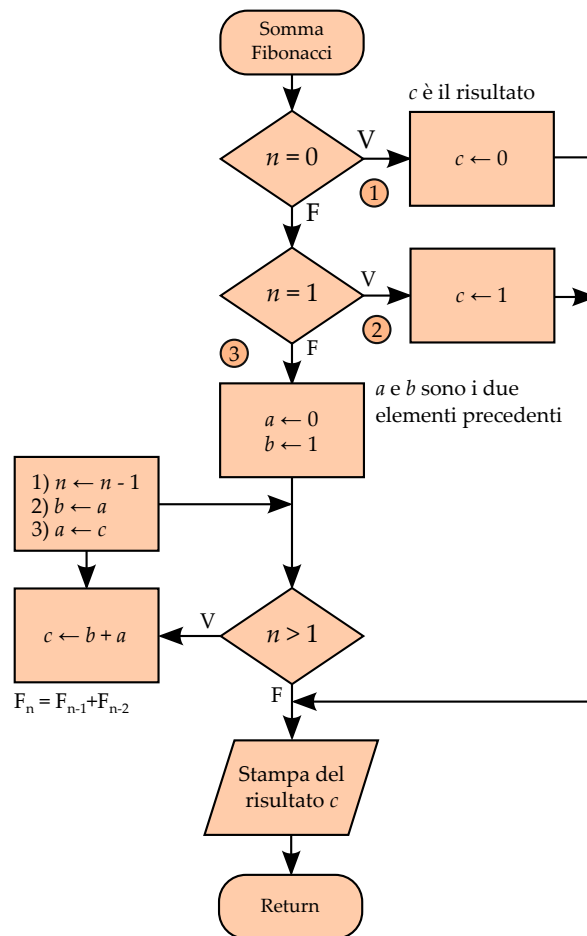



Figura 1.39: Fibonacci: Somma Fibonacci

Alla fine dell'algoritmo è posto il blocco di Output con la stampa del risultato posto nella variabile c.

Si lascia allo studente la verifica della correttezza del diagramma di flusso.

La difficoltà dell'esercizio appena descritto risiedeva in gran parte nella comprensione del testo, espresso in forma non descrittiva o mediante linguaggio naturale, ma attraverso un linguaggio formale e perciò molto asciutto e spesso ostico allo studente. Si propongono di seguito altri esercizi, non risolti, che presentano lo stesso tipo di difficoltà.

1. $\Diamond\Diamond\Diamond$ **Prodotto.** Il prodotto di $x \cdot y$ con $x, y \in \mathbb{N}$ è dato da:

$$x \cdot y = \begin{cases} 0, & \text{se } y = 0; \\ x + x \cdot (y-1), & \text{se } y > 0. \end{cases}$$

   Si fornisca una descrizione in linguaggio naturale delle sole due operazioni aritmetiche di somma e sottrazione da effettuare per calcolare il prodotto di $x \cdot y$ e se ne fornisca il diagramma di flusso. Si noti che l'espressione $x \cdot (y-1)$ è più facile da capire se la si intende come "risultato parziale".
2. $\Diamond\Diamond\Diamond$ **Potenza.** L'elevamento a potenza di x^y con $x \in \mathbb{N}^+$ e $y \in \mathbb{N}$ è dato da:

$$x^y = \begin{cases} 1, & \text{se } y = 0; \\ x \cdot x^{y-1}, & \text{se } y > 0. \end{cases}$$

   Si fornisca una descrizione in linguaggio naturale delle sole quattro operazioni aritmetiche da effettuare per calcolare l'elevamento a potenza di x^y e se ne fornisca il diagramma di flusso. Anche in questo caso l'espressione x^{y-1} è più facile da capire se la si intende come "risultato parziale".
3. $\Diamond\Diamond\Diamond$ **Fattoriale.** Il fattoriale di n (si scrive $n!$) con $n \in \mathbb{N}$ è dato da:

$$n! = \begin{cases} 1, & \text{se } n = 0; \\ n \cdot (n-1)!, & \text{se } n > 0. \end{cases}$$

   Si fornisca una descrizione in linguaggio naturale delle sole quattro operazioni aritmetiche da effettuare per calcolare il fattoriale di n e se ne fornisca il diagramma di flusso. L'espressione $(n-1)!$ è da intendersi come "risultato parziale".
4. $\Diamond\Diamond\Diamond$ **Sommatoria.** La sommatoria dei numeri naturali da 1 a n è descritta matematicamente come

$$S_n = \sum_{a=1}^{n} a$$

   Se ne tracci il diagramma di flusso, ponendo particolare attenzione alla scelta dell'iterazione, e se ne verifichi l'uguaglianza con

$$S_n = \frac{n \cdot (n+1)}{2}$$

5. $\Diamond\Diamond\Diamond$ **Sommatoria 2.** La sommatoria dei numeri naturali da n a m con $m > n$ è descritta matematicamente come

$$S_n = \sum_{a=n}^{m} a$$

   Se ne tracci il diagramma di flusso, ponendo particolare attenzione alla scelta dell'iterazione, e se ne verifichi l'uguaglianza con

$$S_n = \frac{(m-n+1)(m+n)}{2}$$

Vengono ora proposti ulteriori tre esercizi relativi a problemi di una certa complessità. La loro funzione è quella di illustrare allo studente come un qualsiasi problema possa e debba essere scomposto in problemi più semplici da affrontare separatamente. Non vengono forniti esercizi non risolti simili a quelli proposti.

Esercizio - ♢♢♦ L'algoritmo esteso di Euclide

Dati due numeri naturali n e m con $n > m > 0$, si vogliono calcolare due interi a e b con $a, b \in Z$, tali che:

$$an + bm = MCD \tag{1.31}$$

Verranno proposte due soluzioni: una più "informatica" (esaustivamente esposta in KNUTH [8]) ed una più "matematica". La prima si presta facilmente ad una risoluzione mediante il PC, la seconda implica passaggi di natura algebrica che sono, invece, tipici dell'approccio matematico.

Si supponga, a mo' d'esempio, che $n = 1638$ e $m = 533$, come nell'esercizio svolto 1.5. Supponendo l'uso di due ulteriori variabili temporanee a' e b' di una variabile q destinata a contenere il quoziente e di una variabile r destinata al resto, si dà il seguente algoritmo atto al calcolo dei coefficienti della equazione (1.31):

1. Si ponga $a' \leftarrow b \leftarrow 1$ e $b' \leftarrow a \leftarrow 0$
2. Si calcoli il quoziente q e il resto r della divisione di n per m (in tal modo si avrà $n = qm + r$ con $0 < r < m$)
3. Se il resto r vale 0, l'algoritmo termina ed il risultato è $an + bm = MCD$
4. Altrimenti si ponga $n \leftarrow m$, $m \leftarrow r$, $t \leftarrow a'$, $a' \leftarrow a$, $a \leftarrow t - qa$, $t \leftarrow b'$, $b' \leftarrow b$, $b \leftarrow t - qb$
5. Si torni al punto 2.

Si chiede, quindi, di descrivere, mediante diagrammi di flusso, l'algoritmo che permette di trovare i suddetti coefficienti di n e m.

Soluzione

Al punto 3 dell'algoritmo si nota l'uso della variabile temporanea t, utilizzata più per non creare confusione che per reale necessità. Essa non viene comunque utilizzata nella seguente tabella che fornisce il risultato dei calcoli effettuati nell'esecuzione dell'algoritmo, supponendo $n = 1638$ e $m = 533$.

#	n	m	q	r	a'	a	b'	b
01	1638	533	3	39	1	0	0	1
02	533	39	13	26	0	1	1	-3
03	39	26	1	13	1	-13	-3	40
04	26	13	2	0	-13	14	40	-43

Tabella 1.5: Algoritmo esteso di Euclide

Il risultato dell'algoritmo è quindi:

$$14 \cdot 1638 - 43 \cdot 533 = 13 \tag{1.32}$$

L'algoritmo esteso di Euclide trova inaspettate applicazioni. E' usato, ad esempio, nella cosidetta moltiplicazione di Montgomery, che è parte integrante dell'algoritmo RSA usato nella protezione delle carte di credito. Quindi, come si vede, studiare Euclide non è poco moderno o poco attuale.

Adesso che l'algoritmo è stato verificato, si può procedere nell'implementazione del relativo diagramma di flusso. Esso sarà del tutto simile al diagramma di fig. 1.7.

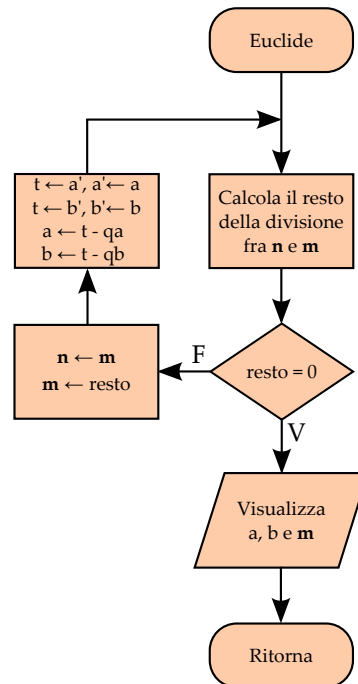



Figura 1.40: Diagramma di flusso dell'algoritmo esteso di Euclide

La differenza evidente con il diagramma di flusso dell'algoritmo di Euclide è data dal blocco che segue l'aggiornamento delle variabili n e m. A parte ciò, il diagramma è identico al precedente.

La particolarità dell'algoritmo, ancora una volta, è la semplicità. Lo stesso diagramma di flusso appare come una semplice estensione dell'algoritmo di Euclide, con la sola aggiunta di un blocco di Elaborazione piuttosto banale.

Più difficile ci appare, piuttosto, la comprensione del ruolo delle variabili a, a', b e b' e ancor meno il motivo per cui esse vengono inizializzate con i valori 0, 1, 1, 0. Si lascia allo studente tale utile esercizio.

Una volta dimostrati questi passaggi, però, l'implementazione appare molto semplice e lineare, sia nella comprensione che nella documentazione e nella successiva stesura del diagramma di flusso.

Un po' più complessa si presenta, invece, la soluzione "matematica", nel senso che la descrizione dell'algoritmo è un po' più elaborata, dato che deve descrivere dei passaggi matematici.

Essa si presenta identica all'algoritmo di Euclide fino alla determinazione del MCD. Se $n = 1638$ e $m = 533$, applicando le regole dell'algoritmo di Euclide si arriva alla ben nota tabella illustrata di seguito:

#	n	m	q	r
01	1638	533	3	39
02	533	39	13	26
03	39	26	1	13
04	26	13	2	0

Tabella 1.6: Risultati parziali dell'algoritmo di Euclide

Siccome in linea generale si ha che:

$$r = n - q \cdot m \tag{1.33}$$

dalla riga 3 della tabella 1.6 si ha:

$$13 = 39 - 26 \tag{1.34}$$

Dalla riga 2 della tabella 1.6 si evince che:

$$26 = 533 - 13 \cdot 39 \tag{1.35}$$

per cui, sostituendo nella 1.34 si ottiene:

$$13 = 39 - (533 - 13 \cdot 39) \tag{1.36}$$

ossia

$$13 = -533 + 14 \cdot 39 \tag{1.37}$$

Dalla riga 1 della tabella 1.6, però, si evince che:

$$39 = 1638 - 3 \cdot 533 \tag{1.38}$$

per cui, sostituendo nella 1.37 si ha:

$$13 = -533 + 14 \cdot (1638 - 3 \cdot 533) \tag{1.39}$$

ossia

$$13 = 14 \cdot 1638 - 43 \cdot 533 \tag{1.40}$$

che è esattamente lo stesso risultato ottenuto nella 1.32.

Dagli illustrati passaggi algebrici si deve poter sintetizzare una regola, possibilmente semplice, che permetta una descrizione precisa e non ambigua dell'algoritmo che si deve eseguire per pervenire al risultato della equazione 1.40.

Una descrizione sommaria, magari racchiusa in un unico blocco di Elaborazione, non permetterebbe ad un ipotetico esecutore di pervenire "meccanicamente" al risultato, per cui si dovrà elaborare un algoritmo dotato di un buon grado di dettaglio.

Tale descrizione non è del tutto banale e lo studente farebbe bene a meditare sulla differenza esistente fra il "saper fare" qualcosa e "saperlo descrivere".

Una prima banale osservazione che si può fare consiste nel suddividere l'intero algoritmo esteso di Euclide in una Sequenza formata da due blocchi di Elaborazione: un primo blocco che potremmo denominare "Euclide", che racchiude, appunto, l'algoritmo di Euclide così come è descritto nel diagramma di fig. 1.7 ed un secondo blocco che potremmo denominare "Algebra" e che racchiude i passaggi algebrici da (1.34) a (1.40), ancora da descrivere. Ciò ci porta ad un primo, sommario diagramma di flusso, come quello illustrato in fig. 1.41.

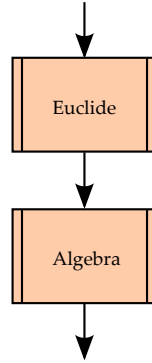



Figura 1.41: Algoritmo esteso di Euclide descritto sommariamente

Una seconda osservazione che si può fare riguarda l'inizio dell'algoritmo. Se, per calcolare il MCD fra due numeri dati sono necessarie p ripetizioni del ciclo di fig. 1.7, e se si pone $a = 1$ e $b = -q_{p-1}$, si potrà scrivere:

$$MCD = r_{p-1} = a \cdot n_{p-1} + b \cdot m_{p-1} \tag{1.41}$$

e siccome m_{p-1} non è altro che r_{p-2}, si potrà scrivere:

$$MCD = r_{p-1} = a \cdot n_{p-1} + b \cdot (n_{p-2} - q_{p-2} \cdot m_{p-2}) \tag{1.42}$$

e constatando che n_{p-1} equivale a m_{p-2}, si avrà:

$$MCD = r_{p-1} = a \cdot m_{p-2} + b \cdot (n_{p-2} - q_{p-2} \cdot m_{p-2}) \tag{1.43}$$

da cui

$$MCD = r_{p-1} = a \cdot m_{p-2} + b \cdot n_{p-2} - b \cdot q_{p-2} \cdot m_{p-2} \tag{1.44}$$

ed infine

$$MCD = r_{p-1} = b \cdot n_{p-2} + (a - b \cdot q_{p-2}) \cdot m_{p-2} \tag{1.45}$$

ed operando uno *swap* dei coefficienti attraverso la variabile muta t, si ha:

$$t \leftarrow b; \quad b \leftarrow a - b \cdot q_{p-2}; \quad a \leftarrow t \tag{1.46}$$

ed infine

$$MCD = r_{p-1} = a \cdot n_{p-2} + b \cdot m_{p-2} \tag{1.47}$$

I passaggi algebrici da (1.41) a (1.47) vanno ripetuti per $p-2$ volte per ottenere i coefficienti a e b definitivi, ma se si rinuncia alla loro eleganza, si possono tutti

condensare nelle tre assegnazioni indicate in 1.46. Si tratta, anche in questo caso di uno *swap* fra a e b attraverso la variabile muta t.

Ad esempio, nel caso trattato nel presente esercizio si contano 4 ripetizioni del ciclo, di cui l'ultima è ininfluente dato che la variabile b viene inizializzata con $-q_{p-1}$. Il primo *swap* viene effettuato a partire dal terzultimo ciclo, quindi si hanno $p-2$ iterazioni.

E' quindi possibile, ora, tracciare un diagramma di flusso del blocco "Algebra", avente la forma di subroutine, come in fig. 1.42:

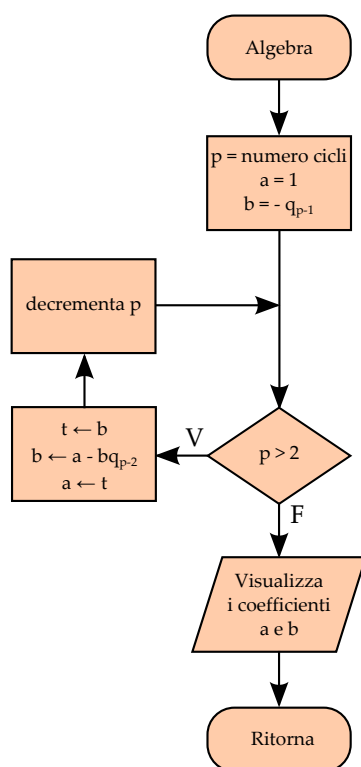



Figura 1.42: Diagramma di flusso della subroutine Algebra

Prima di illustrare il diagramma di flusso si deve sottolineare che si suppone già eseguito l'algoritmo di Euclide e che i valori relativi alla tabella 1.6 si danno per noti.

Il primo blocco rappresenta il blocco di inizializzazione dell'algoritmo, ove vengono inizializzati i valori di a e b. Il valore di a viene inizializzato a 1, mentre b viene inizializzato al valore $-q_{p-1}$. Contestualmente, siccome l'Iterazione che segue è del tipo con numero di cicli definito, viene inizializzato l'indice (o contatore di ciclo) p al valore 4. Tale valore coincide con il numero di iterazioni effettuate nell'algoritmo di Euclide.

Essendo l'Iterazione a test iniziale, viene valutato se il contatore è maggiore di 2, nel qual caso viene effettuato lo *swap* fra a e b. Al termine della Sequenza (riassunta nel presente caso in un unico blocco di Elaborazione) viene decrementato l'indice. Detto ciclo viene ripetuto per $p-2$ volte, al termine delle

quali a e b contengono i valori dei coefficienti che rappresentano il risultato dell'algoritmo.

Esercizio - ♢♦♦ Il problema delle dodici monete

Siano date 12 monete apparentemente identiche, fra le quali potrebbe esservi celata una moneta falsa. La presenza dell'eventuale moneta falsa è determinabile solamente pesandola: essa potrebbe pesare di più o di meno rispetto a quelle autentiche. Si chiede di descrivere un algoritmo che permetta di scoprire l'eventuale moneta falsa, determinando anche se essa pesa di più o di meno rispetto alle altre, utilizzando una bilancia a due piatti ed effettuando non più di tre pesate.

Il problema è mirabilmente trattato in LUCCIO [?].

Soluzione

Innanzitutto è utile tentare di valutare se il problema è teoricamente risolvibile o meno, cercando di quantificare il numero di diversi casi ipotizzabili.

Si potrebbe ipotizzare che la prima moneta sia falsa e più leggera rispetto alle altre, oppure che sia falsa e più pesante delle altre. Analogamente si potrebbe procedere per le restanti 11 monete, per un totale di 24 casi diversi che potremmo nominare 1L (la moneta 1 è falsa e più Leggera delle altre), 1P (la moneta 1 è falsa e più Pesante delle altre), 2L, 2P, 3L 3P, ecc. Infine si potrebbe ipotizzare che nessuna delle 12 monete sia falsa, indicando tale caso con NF (Nessuna moneta Falsa). In totale si avrebbero 25 diversi casi.

Effettuando una pesata con una bilancia a piatti si hanno tre diversi casi:

1. il peso sul piatto di sinistra è minore di quello sul piatto di destra;
2. il peso sul piatto di sinistra è maggiore di quello sul piatto di destra;
3. il peso sul piatto di sinistra è uguale a quello sul piatto di destra.

Effettuando due pesate con lo stesso metodo si hanno $3^2 = 9$ casi differenti ed effettuando tre pesate si hanno $3^3 = 27$ casi differenti, quindi ad una prima superficiale analisi il problema sembra risolvibile con tre pesate.

Si supponga di non avere strategia alcuna e di confrontare due monete prese a caso, ad esempio la moneta 1 e la moneta 2, e di porre la moneta 1 sul piatto di sinistra e la moneta 2 sul piatto di destra. Si possono avere i casi rappresentati in fig. 1.43.

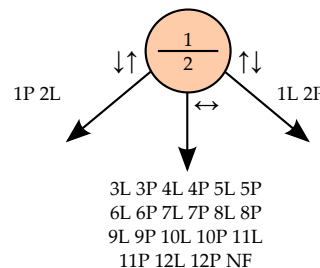



Figura 1.43: Pesata fra due monete

Se i due piatti rimangono in parità si è sicuri che le due monete non sono false, ma potrebbe essere falsa la moneta 3, oppure la 4, la 5, ... Se scende

il piatto di sinistra si possono avere due dstinti casi: la moneta 1 è falsa ed è più pesante rispetto alla altre (1P), oppure la moneta 2 è falsa ed è più leggera rispetto alle altre (2L). Se scende il piatto di destra si possono avere analogamente altri due casi distinti: la moneta 1 è falsa ed è più leggera rispetto alla altre (1L), oppure la moneta 2 è falsa ed è più pesante rispetto alle altre (2P).

Si supponga che scenda il piatto di destra, rendendo possibili i casi 1L e 2P. Sarebbe sufficiente effettuare una seconda pesata utilizzando le monete 1 e 3 (vedi fig. 1.44) per poter stabilire inequivocabilmente qual è la moneta falsa: se ponendo la moneta 1 sul piatto di sinistra e la moneta 3 sul piatto i piatti restano in equilibrio, è evidente che la moneta falsa è la 2 ed è più pesante rispetto alle altre. Se scende il piatto di destra, significa che la moneta falsa è la 1 ed è più leggera rispetto alle altre. Il piatto di sinistra non può scendere, perché questo implicherebbe una seconda moneta falsa, che va contro i dati iniziali del problema.

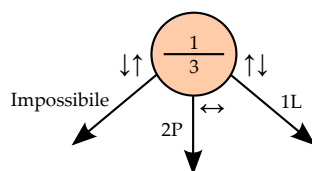



Figura 1.44: Pesata fra due monete (caso 1L 2P)

Una situazione del tutto analoga si ha nel caso 1P e 2L. In entrambi i casi sono sufficienti 2 sole pesate per stabilire quale sia la moneta falsa e se pesa di più o di meno rispetto a quelle autentiche. In fig. 1.45 sono illustrati i due suddetti casi.

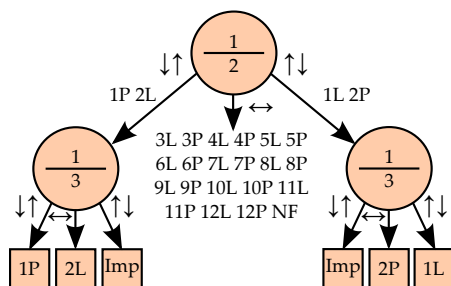



Figura 1.45: Pesata fra due monete (casi 1L 2P e 1P 2L)

Il terzo caso, però, è più complesso. Se i piatti, durante la prima pesata, rimangono in equilibrio, restano possibili (e da esplorare!) ben 21 casi, con sole due pesate a disposizione. Si è già visto che con 2 pesate si possono valutare solo 9 casi distinti, per cui si può concludere che se la prima pesata è effettuata con due monete non si risolve il problema mediante tre pesate, se non si è fortunati. E′ necessario, quindi, un cambio di strategia.

Guardando la fig. 1.43 si nota che i tre rami che rappresentano i tre casi possibili sono fortemente sbilanciati: sui due rami laterali si perviene subito alla soluzione, sul ramo centrale si affacciano 21 casi diversi. Una strategia un

po' più sensata potrebbe essere quella di riequilibrare detti rami, in modo che i possibili casi siano equamente distribuiti.

Tale approccio è possibile se si pesano più di due monete alla volta, passando, ad esempio a 4 (ossia 2+2). Si supponga, ad esempio, di porre le monete 1 e 2 sul piatto di sinistra e le monete 3 e 4 sul piatto di destra. Un diagramma di tale pesata è fornito in fig. 1.46.

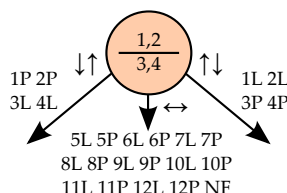



Figura 1.46: Pesata fra quattro monete

La situazione appare visibilmente migliorata, anche se, in caso di parità, i casi da valutare sono ancora 17. Anche in questo caso, due pesate non sono sufficienti.

Redistribuendo ulteriormente i carichi sui tre rami, si può ipotizzare una pesata di 4+4 monete, come illustrato in fig. 1.47.

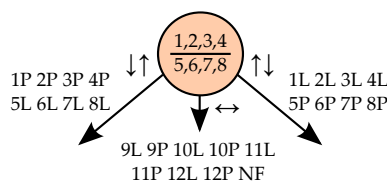



Figura 1.47: Pesata fra otto monete

Ora su ciascun ramo ci sono 9 (o meno) casi da valutare, quindi c'è la possibilità teorica che il problema sia risolvibile con ulteriori due pesate.

La pesata prevede che le monete 1, 2, 3, e 4 siano poste sul piatto di sinistra e che le monete 5, 6, 7 e 8 siano poste sul piatto di destra.

Il piatto di sinistra scende nel caso in cui pesi più del dovuto una fra le monete 1, 2, 3 o 4, oppure nel caso in cui pesi meno del dovuto una fra le monete 5, 6, 7 o 8. In tutto si tratta di 8 casi distinti che, teoricamente, possono essere sondati mediante due pesate ($3^2 > 8$). Analogo ragionamento si può fare se a scendere è il piatto di destra.

Se, invece, i due piatti rstano in equilibrio, si hanno 9 casi possibili: la moneta falsa è una fra le monete 9, 10, 11 o 12 (per un totale di 8 casi distinti), oppure non vi è alcuna moneta falsa fra le dodici. Anche in questo caso due pesate sono sufficienti per sondare i 9 casi ($3^2 \geq 9$).

Ci si può ora concentrare, ad esempio, sul primo dei tre casi (scende il piatto sinistro), continuando ad applicare la stessa strategia, ossia cercando di suddividere gli 8 casi del ramo di sinistra di fig. 1.47 in tre parti più o meno equivalenti, ad esempio 3+2+3. Si tratta, quindi, di stabilire quante monete pesare.

Si scarta subito l'idea di pesare 2+2 monete (ad esempio la 1 e la 2 poste sul piatto di sinistra e la 3 e la 4 poste sul piatto di destra), perché in caso di equilibrio dei piatti si avrebbe solamente una pesata per valutare 4 casi possibili (4L, 5L, 6L e 7L). Quindi si dovrà procedere con una pesata da 3+3 monete, ad esempio ponendo le monete 1, 2, e 3 sul piatto di sinistra e le monete 5, 6 e 7 sul piatto di destra. Anche questa scelta, però, è da scartare subito, perché, come si vede dalla fig. 1.48 i rami sono fortemente sbilanciati.

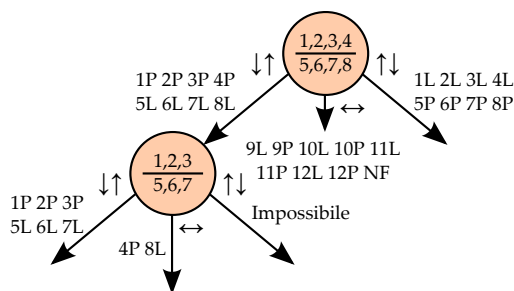



Figura 1.48: Pesata fra otto monete (ciclo 1P 2P 3P e 5L 6L 7L)

Inoltre, ad esempio, pesando i due gruppi di monete 1, 2, 3 e 5, 6, 7 non sarà mai possibile che le monete 5, 6, 7 pesino di più ripetto alle monete 1, 2, 3, dato che nella pesata precedente si era già stabilito che il gruppo 1, 2, 3, 4 pesasse di più rispetto il gruppo 5, 6, 7, 8.

Si potrebbe allora ipotizzare una diversa distribuzione delle monete, come, ad esempio, illustrato in fig. 1.49.

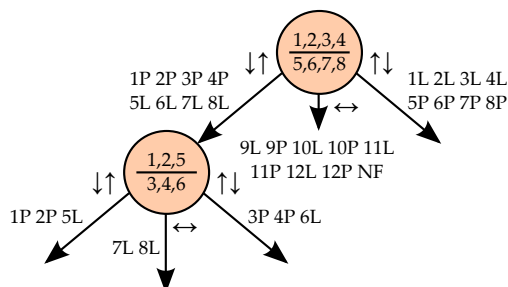



Figura 1.49: Pesata fra otto monete (ciclo 1P 2P 5L e 3P 4P 6L)

Quest'ultima scelta soddisfa le esigenze di una corretta distribuzione sui singoli rami. Il ramo di sinistra prevede 3 casi distinti, risolvibili mediante l'ultima pesata. Il ramo centrale prevede due soli casi, anche loro risolvibili mediante l'ultima pesata, mentre il ramo di destra prevede, simmetricamente, tre casi.

E' bene sottolineare che la scelta delle 3+3 monete è fondamentale. Supponendo di scegliere due gruppi diversi di monete, ad esempio 1, 2, 7 sul piatto di sinistra e 5, 6, 3 sul piatto di destra, si otterrebbe un ramo di sinistra con quattro casi possibili (1P, 2P, 5L e 6L) e due casi sui restanti rami (rispettivamente 4P e 8L sul ramo di centro e 7L e 3P sul ramo di destra).

Fatte queste doverose riflessioni è ora possibile completare questa parte del diagramma delle pesate.

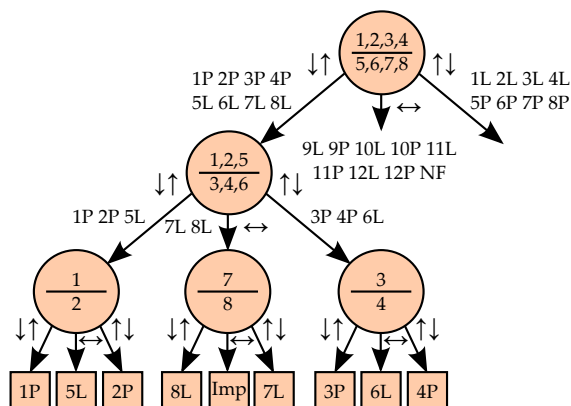



Figura 1.50: Pesata fra otto monete (ramo sinistro completo)

Il diagramma delle pesate del lato destro riflette gli stessi ragionamenti fatti per il ramo sinistro, per cui è possibile illustrarlo senza ulteriori commenti.

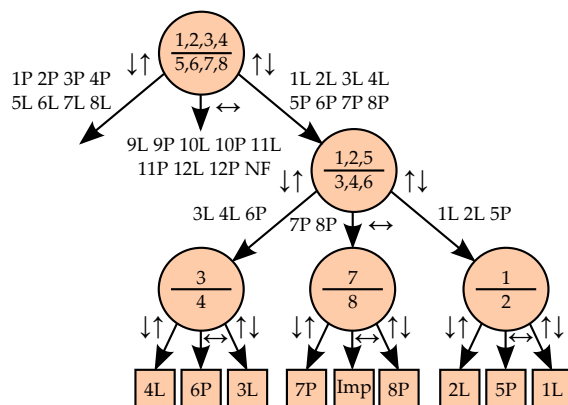



Figura 1.51: Pesata fra otto monete (ramo destro completo)

Il ramo centrale della prima pesata è leggermente diverso dai precedenti due. In caso, dopo la prima pesata, di discesa del piatto di sinistra, infatti, si poteva contare su un totale di 8 monete diverse. Scegliendole accuratamente si è potuto ottenere un ottimale equilibrio dei rami. Ciò non è possibile nella presente situazione, dato che le monete coinvolte sono solo 4.

Supponendo, infatti di confrontare fra loro le monete 9, 10 (sul piatto di sinistra) e 11,12 (sul piatto di destra), nel caso di abbassamento del piatto di sinistra si dovrebbero valutare 4 casi (9P, 10P, 11L, 12L) con una sola pesata, rendendo irrisolvibile il problema. Si rende, quindi, necessario un cambio di strategia.

Una possibile soluzione consiste nel "prendere in prestito" una moneta già utilizzata e di cui si conosce già l'autenticità, senza introdurre quindi altre variabili. Infatti se, dopo la prima pesata i due piatti sono in equilibrio, significa

che le monete 1, 2, 3, 4, 5, 6, 7 e 8 sono sicuramente autentiche e sono pertanto utilizzabili al suddetto fine. Una possibile scelta è illustrata in fig. 1.52, che utilizza, come moneta autentica, la 1.

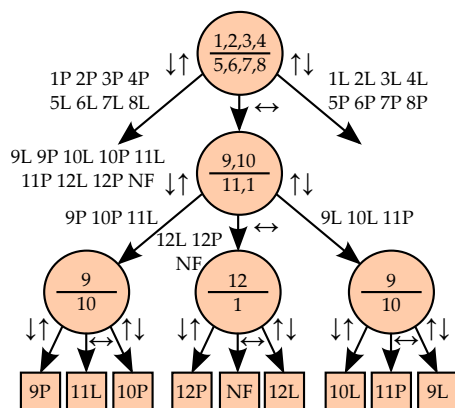



Figura 1.52: Pesata fra otto monete (ramo centrale completo)

A questo punto è possibile riassumere l'intero diagramma delle pesate del problema delle 12 monete, come illustrato in fig. 1.53.

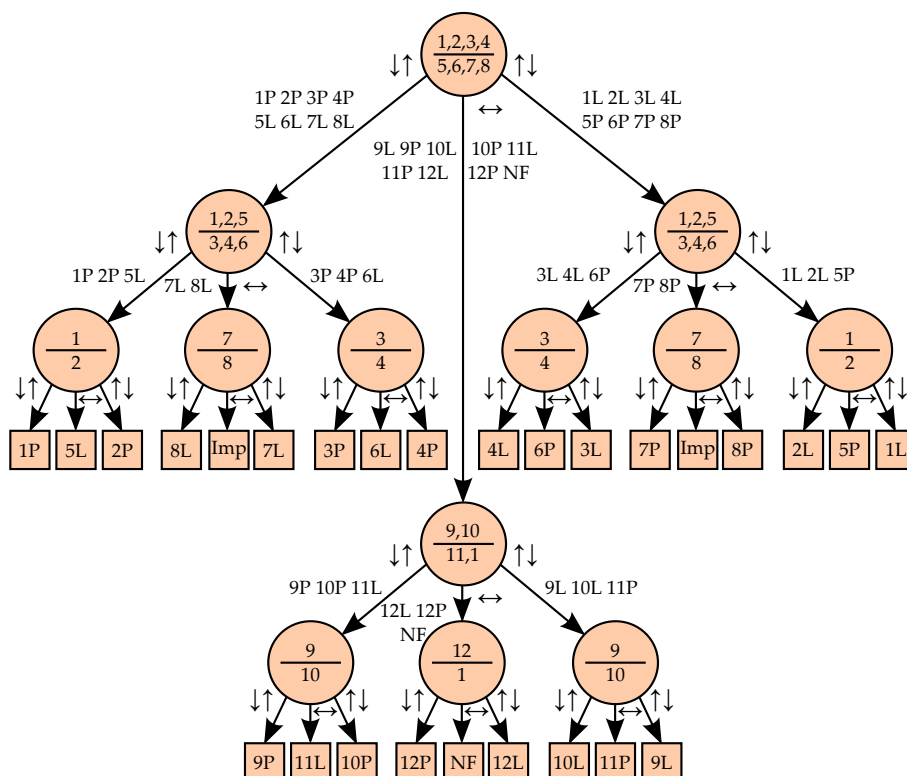



Figura 1.53: Diagramma completo delle pesate del problema delle 12 monete

Il problema così concluso ha presentato qualche difficoltà da superare. Probabilmente i due ostacoli maggiori sono stati rappresentati dal bilanciamento dei singoli rami e dall'uso della moneta "autentica". Non si tratta certamente di difficoltà insormontabili, ma richiede un certo impegno.

Nel primo caso è stata d'aiuto qualche banale riflessione matematica, nel secondo caso è stata necessaria un po' di creatività. La capacità di usare entrambe può essere indubbiamente rafforzata attraverso gli esercizi, lo studio ed il costante tentativo di correlare fra loro le conoscenze acquisite.

E' possibile che lo studente provi un iniziale scoramento nell'affrontare esercizi che richiedono un certo grado di analisi e di astrazione, ma il compito dell'esercizio "difficile" consiste proprio nel far capire all'allievo che il lavoro non è ancora finito e che, anzi, è necessario rinnovato impegno e ulteriori sforzi.

Si lascia allo studente il compito di tracciare il diagramma di flusso dell'algoritmo che risolve il problema delle 12 monete.

Esercizio - ♦♦♦ Calcolo dell'area di un poligono irregolare

Siano dati un insieme n di punti posti sul piano cartesiano, formanti un poligono irregolare di n lati. Siano i valori di ascissa e ordinata dei punti definiti nell'insieme dei numeri relativi. Si chiede la definizione di un algoritmo utile al calcolo dell'area del poligono, alternativo alla formula dell'area di Gauss (vedi formula 1.63 a pag. 88).

Soluzione

Si supponga di aver definito i vertici del poligono di fig. 1.54 che, quindi, risultano essere noti nel loro valore di ascissa e di ordinata.

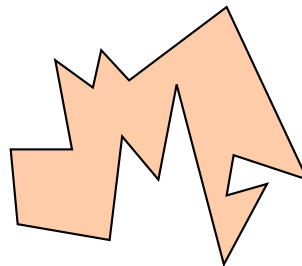



Figura 1.54: Un poligono con n vertici

Uno dei metodi più semplici per calcolare l'area del suddetto poligono consiste nel suddividerlo in tanti triangoli e calcolarne poi separatamente l'area dopo aver calcolato i singoli lati.

A tal fine può risultare utile la formula di Erone (vedi 1.48) che permette di calcolare l'area di un triangolo qualsiasi essendone noti i singoli lati:

$$A_T = \sqrt{p(p-a)(p-b)(p-c)} \tag{1.48}$$

dove a, b e c sono i lati del triangolo e p il semiperimetro. Siccome il calcolo dei singoli lati del triangolo si riduce al calcolo della distanza fra due punti (i vertici), il nocciolo della questione consiste nel trovare un modo semplice e facilmente implementabile sotto forma di algoritmo della suddivisione in triangoli di un poligono qualsiasi.

Si noti, ad esempio, la figura 1.55. Essa è stata suddivisa in triangoli senza un criterio apparente e senza seguire un determinato algoritmo.

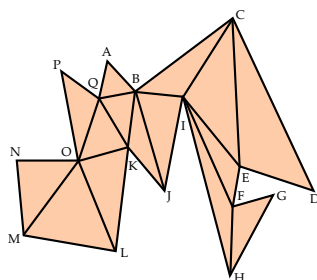



Figura 1.55: Il poligono suddiviso in triangoli

Una volta ottenuta, però, una tale suddivisione, il calcolo dell'area del poligono è piuttosto facile: si deve calcolare la lunghezza dei tre lati di ciascun triangolo ed applicare la formula di Erone per calcolarne l'area. Le singole aree, infine, dovranno essere sommate per ottenere l'area totale.

Si supponga, ora, di voler stabilire un criterio (un algoritmo) per la suddivisione del poligono in triangoli, ad esempio collegando il vertice M con il vertice O, al fine di ottenere il triangolo MNO (si faccia riferimento alla fig. 1.56); collegando, poi, il vertice O con il vertice Q, e così via.

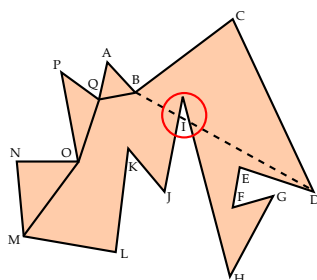



Figura 1.56: Tentativo di suddivisione in triangoli

Tale azione è piuttosto facile da implementare se si pensa ai vertici come ad una successione di punti sul piano: è sufficiente tracciare un segmento che colleghi i vertici v_M e $v_M + 2$. Analogamente si può procedere congiungendo i vertici O e Q, al fine di ottenere il triangolo OPQ, e congiungendo i vertici Q e B del triangolo QAB.

L'area di questi tre triangoli è facilmente calcolabile separatamente ed una volta calcolata, il relativo triangolo può essere tolto dal poligono, riducendolo progressivamente.

Nasce, però, un problema. Continuando con il processo di identificazione dei triangoli, si vorrebbe congiungere il vertice B con il vertice D (si veda a tal proposito la linea tratteggiata in fig. 1.56), ma ciò non è lecito, dato che il vertice I cadrebbe all'interno dell'area del triangolo BCD. Il calcolo dell'area del triangolo BCD risulterebbe inesatto.

Si potrebbe allora tentare di iniziare da un altro vertice, ad esempio da A, procedendo esattamente come indicato precedentemente.

In tal modo ci si ritroverebbe, però, di fronte ad un altro problema: congiungendo il vertice A con il vertice C, si identificherebbe un triangolo (il triangolo ABC) la cui area non appartiene al poligono, ma è esterna ad esso (si veda la fig. 1.57).

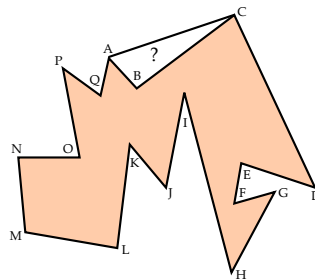



Figura 1.57: Altro tentativo di suddivisione in triangoli

Quindi la suddivisione in triangoli dell'area del poligono non è così immediata come forse poteva sembrare ad un primo momento. Anzi: si dovranno tener ben presenti i due suddetti problemi quando si suddividerà in triangoli il poligono.

Si propone, allora, un approccio leggermente diverso al problema, iniziando col determinare da quale vertice si debba iniziare il suddetto processo. Per semplicità si potrebbe iniziare dal vertice avente ordinata massima (nel presente caso il vertice C). Il motivo di tale scelta si spiega nel seguente modo.

Si immagini il vertice in questione posizionato all'origine di un sistema di assi cartesiani, come visualizzato in fig. 1.58.

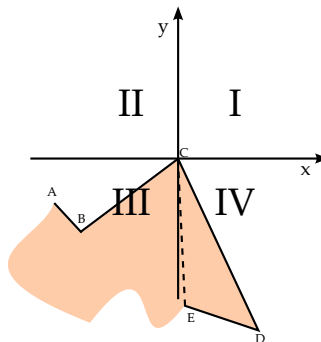



Figura 1.58: Sistema di assi cartesiani con origine in C

Tale situazione è privilegiata rispetto alle altre: i lati CB e CD possono estendersi solamente nei quadranti III e/o IV, oppure possono essere coincidenti con l'ascissa, essendo il vertice C quello con ordinata maggiore. Non potranno, però, mai estendersi nei I o II quadrante con coefficiente angolare diverso da 0.

E' quindi possibile valutare concretamente il caso prospettato in fig. 1.57. Il segmento di congiunzione (nel caso presente il segmento CE. Si veda la figura 1.58) deve formare un angolo, con l'asse delle ascisse, superiore a quello formato dal lato CB ed inferiore a quello formato dal lato CD.

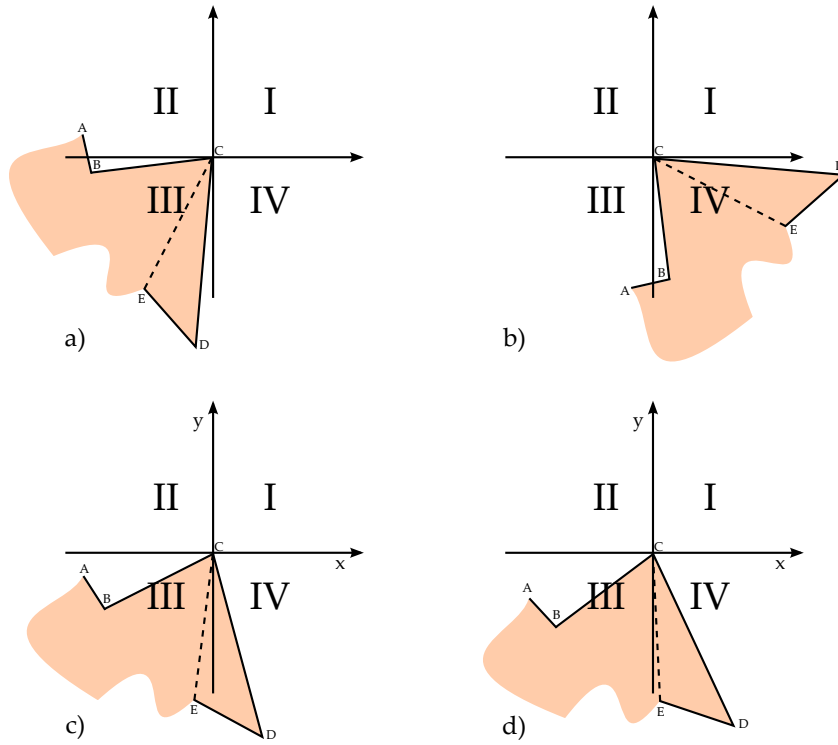



Figura 1.59: Quattro possibili casi

Detta regola deve, però, essere valida sempre. Per verificarla si analizzino i quattro possibili casi visualizzati in fig. 1.59.

Nel **caso illustrato in fig. 1.59a** sia il lato CB che il lato CD si estendono nel III quadrante, con

$$m_{CB} < m_{CD} \tag{1.49}$$

dove m_{CB} è il coefficiente angolare (positivo) del segmento CB e m_{CD} è il coefficiente angolare (positivo) del segmento CD, ovvero

$$m_{CB} = \frac{B_y - C_y}{B_x - C_x} \quad e \quad m_{CD} = \frac{B_y - D_y}{B_x - D_x} \tag{1.50}$$

Quindi, se m_{CE} è il coefficiente angolare del segmento di congiunzione CE, affinché il triangolo BDE rappresenti un triangolo appartenente al poligono dato si dovrà avere che

$$m_{CB} < m_{CE} \tag{1.51}$$

e

$$m_{CE} < m_{CD} \tag{1.52}$$

ovvero, in conclusione,

$$m_{CB} < m_{CE} < m_{CD} \tag{1.53}$$

che è la condizione necessaria affinché il triangolo appartenza al poligono.

Nel **caso illustrato in fig. 1.59b** sia il lato CB che il lato CD si estendono nel IV quadrante, con

$$m_{CB} < m_{CD} \tag{1.54}$$

Si noti che entrambi i coefficienti angolari sono sempre negativi, a meno dell'unica eccezione in cui il lato CD ha pendenza 0, essendo coincidente con l'ascissa. Quindi anche in questo caso deve essere verificata la condizione 1.53.

I **casi illustrati in fig. 1.59c e d** sono un po' più complessi, dato che il lato CB si estende nel III quadrante ed il lato CD nel IV quadrante. Ciò implica che il coefficiente angolare di CD è nullo o negativo, mentre quello di CB è nullo o positivo. Devono essere quindi soddisfatte le seguenti due condizioni

$$m_{CE} \begin{cases} < m_{CD}, & \text{se } CE \in III\ Quadrante, \\ > m_{CB}, & \text{se } CE \in IV\ Quadrante, \end{cases}$$

che equivale a scrivere la 1.53.

Quindi, generalizzando, se è verificata la condizione (1.53) si è sicuri che il triangolo CDE appartiene sicuramente al poligono dato.

Si può tentare ora di generalizzare ulteriormente la (1.53), senza doverla limitare al fatto che il vertice C sia quello con ordinata maggiore.

I casi da esaminare con attenzione sono sostanzialmente due:

- il vertice in esame fa capo a due lati che si espandono entrambi nello stesso quadrante;
- il vertice in esame fa capo a due lati che si espandono in due quadranti differenti.

Il primo caso si risolve banalmente con la (1.53), indipendentemente dal numero di quadrante (si lascia la verifica allo studente).

Il secondo caso è leggermente più complesso, ma rimane comunque di semplice soluzione. Si procede nel seguente modo:

- si identificano i quadranti entro i quali si estendono i tre segmenti;
- si verifica se il segmento di congiunzione si estende nello stesso quadrante di uno dei due lati;
- se si, si verifica che sia valida la (1.51) oppure (1.52), a seconda del lato coinvolto;
- altrimenti si esegue una semplice comparazione dei quadranti, si verifica cioè che $Q_{CB} < Q_{CE} < Q_{CD}$.

Si è ora pronti per generalizzare definitivamente la procedura di suddivisione in triangoli del poligono ma, prima, conviene esaminare il secondo problema al quale si è accennato, ovvero l'intersezione di due segmenti (vedi fig. 1.56).

Esistoni vari algoritmi che offrono una valida soluzione al problema. Nel presente esercizio si propone un algoritmo semplice ed estremamente funzionale tratto da SEDGEWICK [10]. Si sottolinea subito esso non è il più funzionale al problema dato: molto più adatto sarebbe stato l'algoritmo denominato *plane sweep* che, però, è più complesso e richiede un grado di astrazione superiore.

In fig. 1.60 sono illustrate le possibili disposizioni di due ipotetici segmenti.

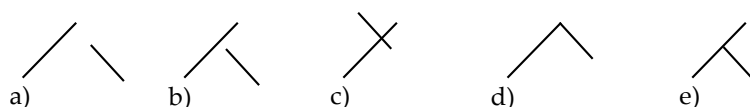



Figura 1.60: Diverse posizioni relative di due segmenti

Il caso illustrato in fig. 1.60a è il più semplice. Infatti, le proiezioni sull'asse X dei due segmenti non si sovrappongono, quindi non è nemmeno possibile che i due segmenti siano intersecanti. I restanti quattro casi sono più complessi.

Il modo più semplice consiste nel ricavare dai punti le equazioni delle rette sulle quali giacciono i segmenti e metterle a sistema al fine di trovare l'eventuale punto di intersezione. Nel caso in cui tale punto esista (ovvero nel caso in cui i due coefficienti angolari siano diversi, dato che possiamo escludere la sovrapposizione dei due segmenti), si dovrà verificare che detto punto sia compreso fra gli estremi del segmento. Ciò può essere agevolmente fatto valutando se le proiezioni del punto cadono fuori dalle proiezioni dei singoli segmenti.

Questo metodo, però, è piuttosto prolisso e richiede la valutazione anche di alcuni casi particolari (coincidenza dei coefficienti angolari, caso di segmenti verticali, segmenti che, pur non sovrapponendosi, giacciono sulla stessa retta, ecc.) che lo rendono ulteriormente farraginoso.

Il metodo proposto nell'opera di SEDGEWICK è proceduralmente semplicissimo, ma concettualmente di non immediata comprensione.

Il cuore dell'algoritmo consiste nel valutare se, dati tre punti, passando da un punto a quello successivo si procede in senso orario o in senso antiorario. In fig. 1.61 sono illustrate le due situazioni. Per passare da P_0 a P_1 e a P_2 si segue, in fig. 1.61a, un andamento orario. Viceversa, per passare da P_0 a P_1 e a P_2 in fig. 1.61b si segue un andamento antiorario.

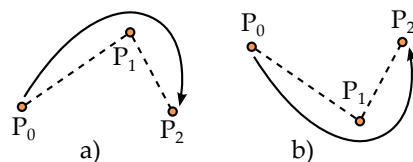



Figura 1.61: Andamento orario e antiorario

Tale valutazione diventa possibile se si confrontano i coefficienti angolari dei segmenti $\overline{P_0P_1}$ e $\overline{P_0P_2}$. Se i tre punti sono percorsi in senso orario il coefficiente angolare del segmento $\overline{P_0P_1}$ deve essere maggiore di quello relativo al segmento $\overline{P_0P_2}$. Altrimenti deve essere minore.

Questa osservazione torna estremamente utile se applicata a due segmenti dei quali si vuole valutare l'eventuale intersezione. Siano dati due segmenti g e h che si intersecano, come evidenziato in fig. 1.62. Se passando dagli estremi del segmento g agli estremi del segmento h si cambia verso (figg. 1.62a e 1.62b) e se ciò avviene anche passando dagli estremi del segmento h agli estremi del segmento g (figg. 1.62c e 1.62d), allora g e h si intersecano.

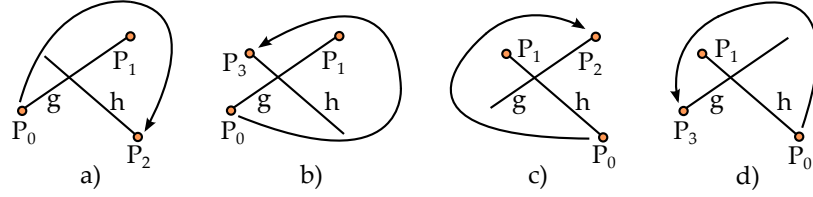



Figura 1.62: Valutazione dell'intersezione fra due segmenti

Ora è possibile generalizzare definitivamente l'algoritmo di scomposizione in triangoli del poligono. Un buon approccio consiste nel definire innanzi tutto gli algoritmi di base che verranno usati successivamente:

- determinazione del coefficiente angolare del segmento (Coefficiente);
- determinazione del senso di percorrenza di tre punti (Percorrenza);
- determinazione dell'intersezione di due segmenti (Intersezione).

Coefficiente. Detto algoritmo ha il compito di confrontare due coefficienti angolari m_1 e m_2 e restituire un valore secondo la seguente funzione:

$$f_m = \begin{cases} +1, & \text{se } m_1 > m_2, \\ 0, & \text{se } m_1 = m_2, \\ -1, & \text{se } m_1 < m_2. \end{cases}$$

Il calcolo del coefficiente angolare, però, non è del tutto privo di ostacoli. Non è infatti sufficiente applicare la (1.50) dato che il segmento in questione potrebbe essere verticale e quindi dare luogo ad una divisione per zero durante il calcolo. Il coefficiente angolare dovrebbe, quindi, essere infinito, ma ciò non è rappresentabile mediante un numero.

Detto problema si risolve facilmente evitando la comparazione

$$\frac{\Delta Y_1}{\Delta X_1} > \frac{\Delta Y_2}{\Delta X_2} \tag{1.55}$$

sostituendola con la seguente

$$\Delta Y_1 \cdot \Delta X_2 > \Delta Y_2 \cdot \Delta X_1 \tag{1.56}$$

e analogamente, invece di

$$\frac{\Delta Y_1}{\Delta X_1} < \frac{\Delta Y_2}{\Delta X_2} \tag{1.57}$$

si può scrivere

$$\Delta Y_1 \cdot \Delta X_2 < \Delta Y_2 \cdot \Delta X_1 \tag{1.58}$$

L'unica condizione da porre affinché le suddette equazioni siano vere riguarda la lunghezza del segmento, che deve essere maggiore di 0.

E' ora possibile tracciare un diagramma di flusso riassuntivo di quanto fin qui esposto.

E' sufficiente, infatti, eseguire i due calcoli indicati dalla (1.56) e dalla (1.58) e, nel caso in cui non risulti vera né l'una né l'altra, dedurre che i due segmenti hanno lo stesso coefficiente angolare.

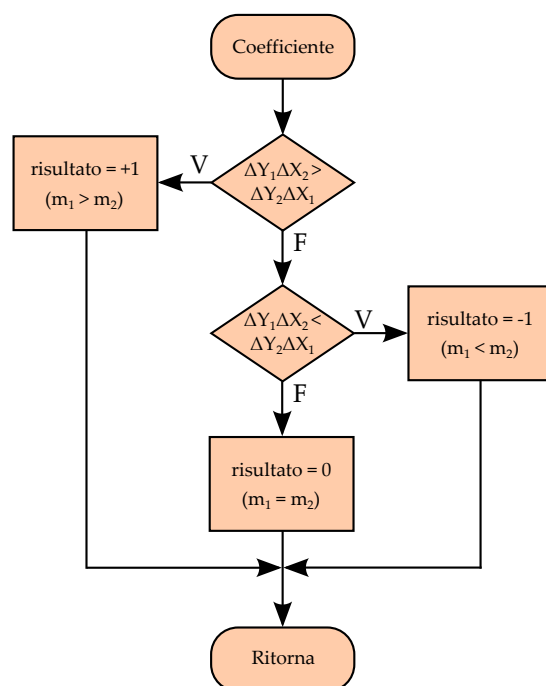



Figura 1.63: Diagramma di flusso della comparazione dei coefficienti angolari

Percorrenza. L'algoritmo di percorrenza ha il compito di valutare se, dati tre punti, passando dall'uno all'altro essi sono percorsi in senso orario o antiorario (vedi fig. 1.61). Esso, inoltre, esamina anche i casi in cui i tre punti siano posti sulla stessa retta ed in quale sequenza. Esaminare detto caso è estremamente importante per identificare le situazioni illustrate nelle figg. 1.60d e 1.60e, per cui tale calcolo torna utile nella valutazione dell'intersezione fra due segmenti.

Più precisamente, vale il seguente specchietto:

- se P_0 è posto fra P_1 e P_2, allora si ritorna -1;
- se P_1 è posto fra P_0 e P_2, allora si ritorna +1;
- se P_2 è posto fra P_0 e P_1, allora si ritorna 0;

L'appartenenza al primo punto emerge dalla valutazione delle seguenti disequazioni:

$$\Delta X_1 \cdot \Delta X_2 < 0 \quad \vee \quad \Delta Y_1 \cdot \Delta Y_2 < 0 \tag{1.59}$$

dove $\Delta X_1 = P_{x1} - P_{x0}$, $\Delta X_2 = P_{x2} - P_{x0}$ e $\Delta Y_1 = P_{y1} - P_{y0}$, $\Delta Y_2 = P_{y2} - P_{y0}$.

L'appartenenza, invece, al secondo punto emerge dalla valutazione della seguente disequazione:

$$\Delta X_1 \cdot \Delta X_1 + \Delta Y_1 \cdot \Delta Y_1 < \Delta X_2 \cdot \Delta X_2 + \Delta Y_2 \cdot \Delta Y_2 \tag{1.60}$$

mentre l'ultimo punto è dato per esclusione dei precedenti due.

La 1.59 mira a valutare il segno della proiezione dei due segmenti per verificare la posizione di P_0, mentre la 1.60 è l'applicazione del teorema di Pitagora, al fine di determinare la lunghezza dei segmenti e quindi valutare la veridicità del secondo punto dello specchietto.

In termini di diagramma di flusso si ha:

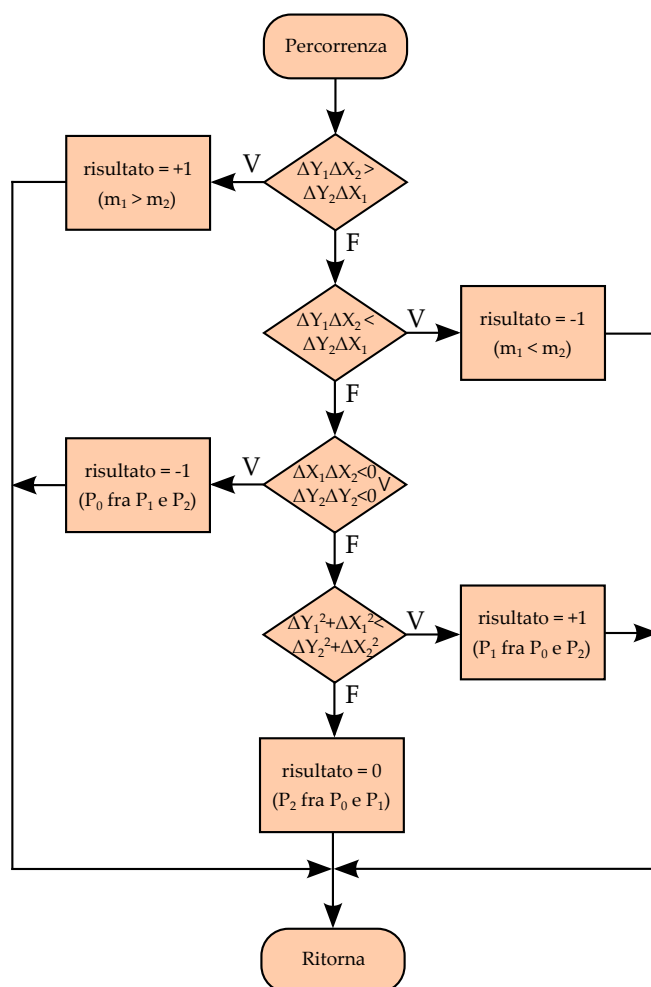



Figura 1.64: Diagramma di flusso del senso di percorrenza di tre punti

Intersezione. L'algoritmo di intersezione ha il compito di valutare se due segmenti si intersecano oppure no. Come evidenziato in fig. 1.62, si ha intersezione da parte di due segmenti se passando dagli estremi del primo segmento agli estremi del secondo si cambia verso e se ciò avviene anche passando dagli

estremi del secondo segmento agli estremi del primo. Detto algoritmo coinvolge, quindi, anche l'algoritmo Percorrenza, come evidenziato nel seguente diagramma di flusso:

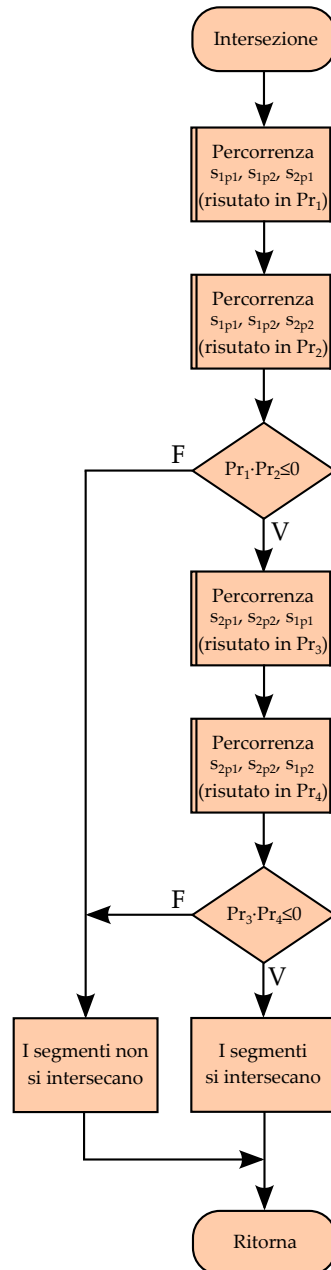



Figura 1.65: Diagramma di flusso dell'intersezione di due segmenti

Ora che gli algoritmi fondamentali sono stati definiti, è possibile generalizzare definitivamente l'algoritmo di scomposizione in triangoli del poligono.

Esso verrà esposto dapprima sotto forma di linguaggio naturale, cercando di limitare le ambiguità e le mancanze di dettagli, e poi mediante diagramma di flusso.

1. si azzera la variabile `Area` che conterrà l'area del poligono.

2. si determina il vertice dal quale iniziare la scomposizione. Visto che si è appurato che un vertice vale l'altro e che essi, dal punto di vista pratico, sono memorizzati in un vettore di punti, conviene eleggere come punto di inizio il primo vertice memorizzato nel vettore, ossia il vertice A di fig. 1.66a. Si pone quindi $VerticeCorrente = PrimoVertice$.

3. si individua il vertice $VerticeCorrente + 2$.

4. si individua il segmento *SegmentoCorrente* i cui estremi sono rappresentati dai punti *VerticeCorrente* e $VerticeCorrente + 2$.

5. si individua il triangolo *TriangoloCorrente* avente vertici *VerticeCorrente*, $VerticeCorrente + 1$ e $VerticeCorrente + 2$.

6. se $VerticeCorrente + 3 = VerticeCorrente$, si va al punto 10.

7. si valuta se è verificata la disequazione

$$m_p < m_c \tag{1.61}$$

dove m_p è il coefficiente angolare del segmento che congiunge il vertice corrente con quello precedente e m_c è il coefficiente angolare del segmento che congiunge il vertice corrente con $VerticeCorrente + 2$. Tale verifica è fatta utilizzando l'algoritmo `Coefficiente`. Se la (1.61) non è verificata si va al punto 13.

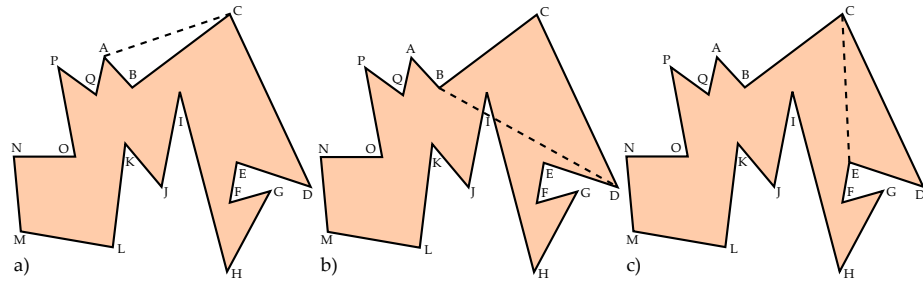



Figura 1.66: Suddivisione in triangoli (individuazione)

8. si valuta se è verificata la disequazione

$$m_c < m_s \tag{1.62}$$

dove m_s è il coefficiente angolare del segmento che congiunge il vertice corrente con quello successivo e m_c è il coefficiente angolare del segmento che congiunge il vertice corrente con $VerticeCorrente + 2$. Tale verifica è fatta utilizzando l'algoritmo `Coefficiente`. Se la (1.62) non è verificata si va al punto 13.

9. si valuta l'intersezione del *SegmentoCorrente* con tutti i lati del poligono. Se almeno una intersezione è presente, si va al punto 13.
10. si calcola la lunghezza di ciascun lato del *TriangoloCorrente*.
11. si calcola l'area del *TriangoloCorrente* mediante la formula di Erone e la si somma ad *Area*.
12. si "toglie" il *TriangoloCorrente* dal poligono, sopprimendo dal vettore dei vertici il vertice $VerticeCorrente + 1$ e si diminuisce il numero di vertici del poligono di 1.
13. se $VerticeCorrente < UltimoVertice$ si incrementa *VerticeCorrente*, altrimenti si pone $VerticeCorrente = PrimoVertice$.
14. se il numero dei vertici del poligono è maggiore o uguale a 3, si torna al punto 3.
15. l'algoritmo è terminato e il risultato è posto in *Area*.

Ora si può iniziare a definire un primo, grossolano, diagramma di flusso complessivo aggiungendo poi via via dei particolari fino a raggiungere un grado di definizione che l'autore ritiene sufficiente.

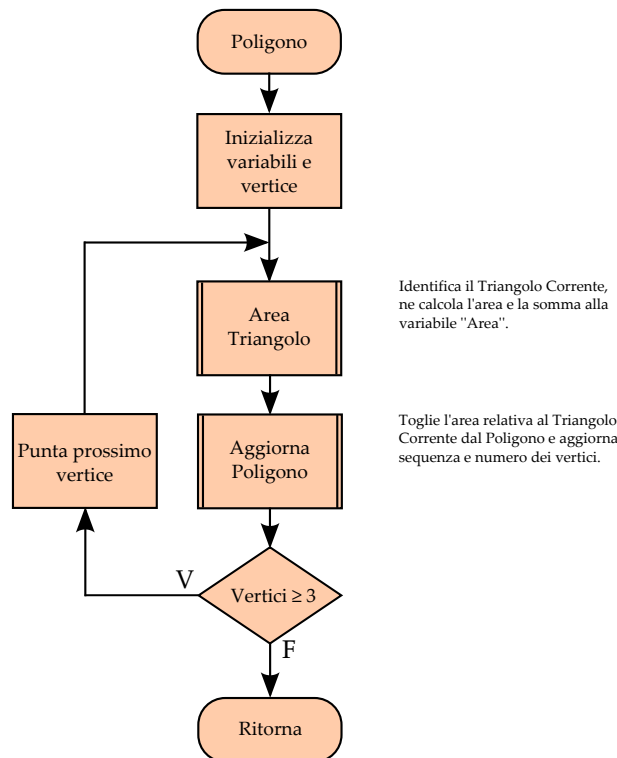



Figura 1.67: Diagramma di flusso di calcolo dell'area del poligono

Il diagramma di fig. 1.67 non è molto dettagliato, ma questo è un pregio, perché accompagna il lettore verso livelli di dettaglio superiori in modo graduale.

Dapprima vengono inizializzate le variabili d'uso, con particolare riferimento ad *Area*, *VerticeCorrente*, *SegmentoCorrente* e *TriangoloCorrente*.

Successivamente si identifica il triangolo formato dai vertici *VerticeCorrente*, $VerticeCorrente + 1$ e $VerticeCorrente + 2$ e se ne calcola l'area mediante la formula di Erone. Detta area viene aggiunta alla variabile *Area*. Tali azioni sono eseguite in "Area Triangolo".

Una volta calcolata detta area essa va sottratta all'area del poligono, togliendo $VerticeCorrente + 1$ dall'elenco dei vertici. Queste azioni, invece, sono eseguite in "Aggiorna Poligono".

Il blocco di elaborazione "Area Triangolo" può essere ulteriormente dettagliato, come evidenziato in fig. 1.68

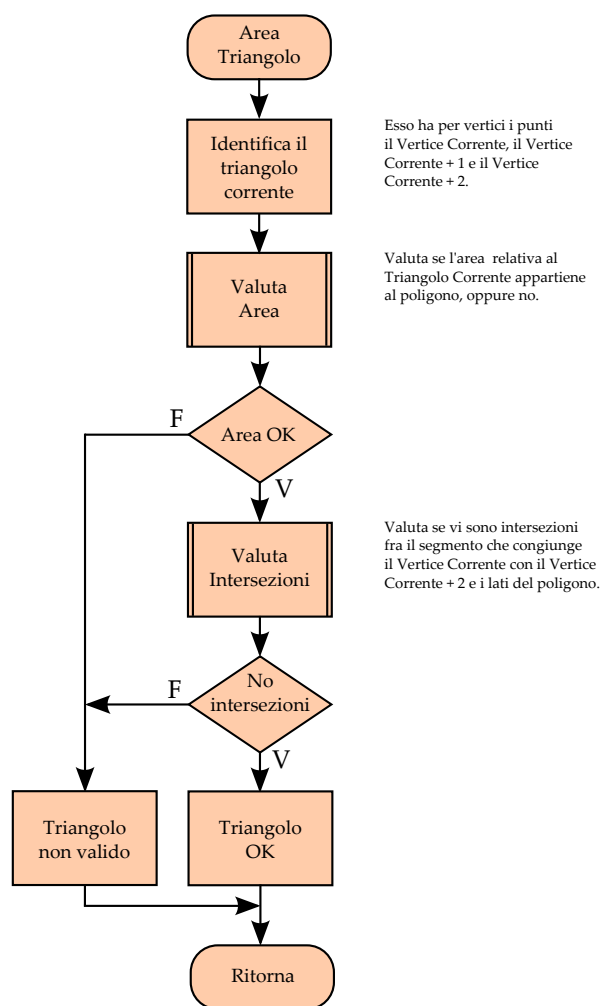



Figura 1.68: Diagramma di flusso di calcolo dell'area del triangolo

Inizialmente viene identificato il Triangolo Corrente, ovvero il triangolo formato dai vertici *VerticeCorrente*, $VerticeCorrente + 1$ e $VerticeCorrente + 2$. Il secondo passo consiste nel valutare se l'area del triangolo corrente fa parte o meno dell'area del poligono, al fine di escludere i casi rappresentati dalla

fig. 1.66a. Nel caso in cui l'area del triangolo corrente non appartenesse al poligono, si dovrebbe uscire con indicazione di triangolo non valido.

La seconda valutazione è relativa ad eventuali intersezioni del Segmento Corrente (ossia il lato del Triangolo Corrente che unisce i vertici *VerticeCorrente* e $VerticeCorrente + 2$) con i lati del poligono. Se uno qualsiasi dei lati del poligono interseca il Segmento Corrente si realizza il caso di fig. 1.66b e si esce dall'algoritmo con indicazione di triangolo non valido.

Se si superano entrambi i test si esce, invece, con l'indicazione che il Triangolo Corrente è valido.

La valutazione del fatto che l'area del Triangolo Corrente appartenga o meno al poligono è rappresentabile mediante il diagramma di fig. 1.69.

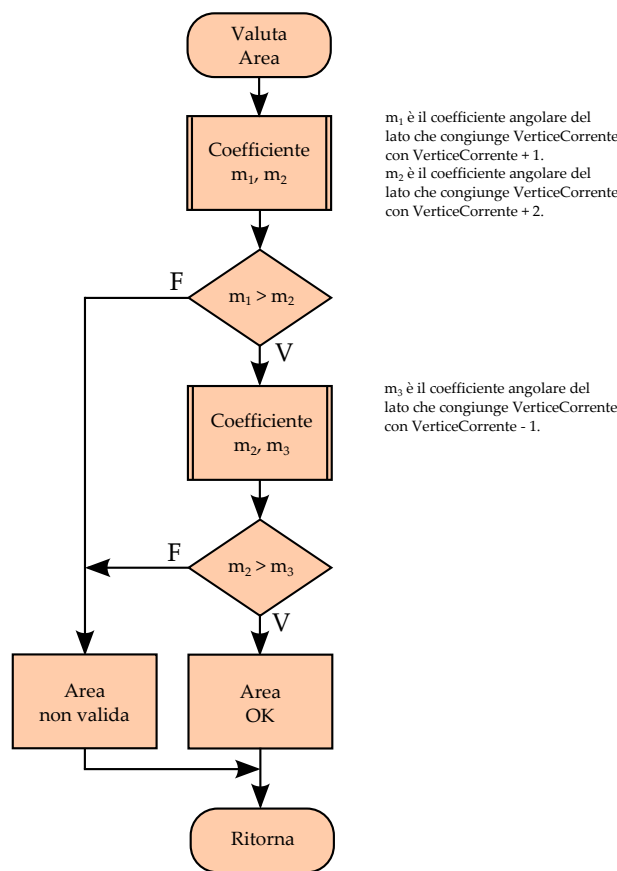



Figura 1.69: Diagramma di flusso di valutazione dell'area del triangolo

L'algoritmo "Valuta Area" utilizza, a sua volta, l'algoritmo fondamentale "Coefficiente", che confronta due coefficienti angolari indicandone il maggiore (vedi il diagramma di fig. 1.63). Con riferimento alla fig. 1.66c, detto algoritmo verifica che sia vera la relazione 1.53.

Se è vera, si esce dall'algoritmo con indicazione di area valida, altrimenti si esce con indicazione di area non valida.

Rimane da valutare se il Segmento Corrente interseca qualche lato del poligono. Detta valutazione è rappresentata dal diagramma di fig. 1.70.

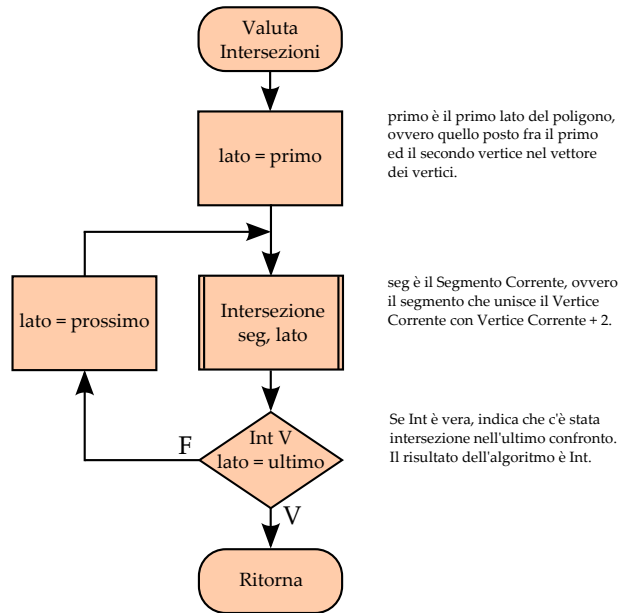



Figura 1.70: Diagramma di flusso di valutazione delle intersezioni

L'algoritmo di fig. 1.70 è piuttosto inefficiente. Come già accennato, vi sono algoritmi, come ad esempio il *plane sweep*, che sono molto più efficienti anche se leggermente più complessi.

L'algoritmo illustrato valuta semplicemente l'intersezione del Segmento Corrente con tutti i lati del poligono, finché non incontra una effettiva intersezione oppure si è giunti all'ultimo lato del poligono. A tal fine è usato l'algoritmo "Intersezione".

Rimane da illustrare un ultimo diagramma, identificato come "Aggiorna Poligono" in fig. 1.67. Detto algoritmo ha il compito di "togliere" dal poligono il Triangolo Corrente, come illustrato in fig. 1.71 e di sommarne l'area ad *Area*.

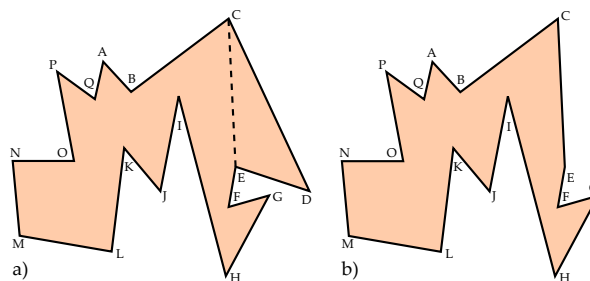



Figura 1.71: Suddivisione in triangoli (aggiornamento)

In tal modo il poligono viene ridotto di un vertice e quindi reso più semplice. Togliendo via via triangoli dal poligono lo si riduce fino ad ottenere un poligono formato da soli tre lati, ovvero l'ultimo triangolo. Detto aggiornamento è rappresentato nel diagramma di fig. 1.72.

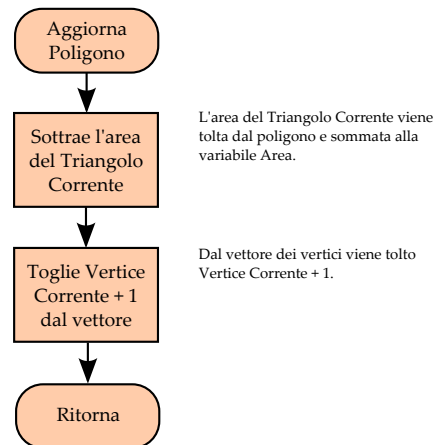



Figura 1.72: Diagramma di flusso di aggiornamento del poligono

Il problema nel suo complesso è stato in tal modo risolto senza entrare eccessivamente nei dettagli ma, anche, senza restare troppo in superficie del problema. Le singole difficoltà sono state affrontate e risolte, anche se non sempre nella maniera più efficiente, nel modo più semplice possibile.

Alcuni algoritmi sono decisamente semplici, mentre altri possono apparire un po' più complessi. In entrambi i casi vanno affrontati con impegno e concentrazione, evitando di sottovalutare gli uni e di sovrastimare gli altri.

Nelle prossime pagine sono presentati una serie di esercizi che permettono allo studente di rafforzare dette qualità.

1.12 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 1. Le soluzioni degli esercizi proposti sono riportate in Appendice A.

Algoritmi

1. ◊◊◊ Si fornisca una definizione personale di algoritmo.
2. ◊◊◊ Si elenchino e si commentino le 5 caratteristiche degli algoritmi.
3. ◊◊◊ Si fornisca un esempio di algoritmo in cui la caratteristica di finitezza è rispettata ed un esempio in cui non è rispettata.
4. ◊◊◊ Si fornisca un esempio di algoritmo in cui la caratteristica di definizione è rispettata ed un esempio in cui non è rispettata.
5. ◊◊◊ Si fornisca un esempio di algoritmo in cui la caratteristica di finitezza è rispettata ed un esempio in cui non è rispettata.
6. ◊◊◊ Si fornisca un esempio di algoritmo avente dei dati di ingresso enumerabili ed un esempio in cui ciò non avviene.
7. ◊◊◊ Si fornisca un esempio di algoritmo che produce dei dati di uscita ed un esempio in cui ciò non avviene.
8. ◊◊◊ Si fornisca un esempio di algoritmo in cui la caratteristica di effettività è rispettata ed un esempio in cui non è rispettata.
9. ◊◊◊ Si illustrino le tre strutture fondamentali degli algoritmi.
10. ◊◊◊ Si spieghi perché la Selezione non è un ciclo.
11. ◊◊◊ Si illustrino le differenze fra i tre tipi di Iterazione.
12. ◊◊◊ Si illustrino tre casi in cui si deve usare l'Iterazione a numero di cicli definito.
13. ◊◊◊ Si illustrino tre casi in cui si deve usare l'Iterazione a numero di cicli non definito e Scelta iniziale.
14. ◊◊◊ Si illustrino tre casi in cui si deve usare l'Iterazione a numero di cicli non definito e Scelta finale.
15. ◊◊◊ Si illustri mediante linguaggio naturale l'algoritmo di Euclide, commentando adeguatamente le parti ritenute ambigue o poco precise.
16. ◊◊◊ Si esegua l'algoritmo di Euclide con le seguenti coppie di numeri: 3699 e 489, 4352 e 442, 5838 e 562, 481 e 2730.
17. ◊◊◊ Si illustri mediante linguaggio naturale l'algoritmo di divisione intera, commentando adeguatamente le parti ritenute ambigue o poco precise.

18. ◊◊◊ "Pierino mangia e beve finchè non ha finito la birra e la pizza". Tale frase, espressa in linguaggio naturale, si presta a diverse interpretazioni. Se ne indichi qualcuna.

19. ◊◊◊ Se la frase dell'esercizio precedente fosse un algoritmo, sarebbero rispettate le 5 caratteristiche proprie degli algoritmi? Quali si? Quali no?

20. ◊◊◊ Nella frase "Pierino mangia e beve finchè non ha finito la birra e la pizza" è presente una mancanza di *effectiveness*, che rende ineseguibili le azioni così come sono descritte. Di quale mancanza si tratta?

21. ◊◊◊ Si ridefinisca la frase dell'esercizio precedente in modo tale da trasformarla in algoritmo.

22. ◊◊♦ "Pierino beve un bicchiere d'acqua". Anche in questo caso l'azione lamenta una mancanza di *effectiveness*. Quale?

23. ◊◊◊ Si ridefinisca la frase precedente in linguaggio naturale in modo che soddisfi le 5 caratteristiche proprie degli algoritmi.

24. ◊◊◊ Si ridefinisca la suddetta frase dettagliandola in maniera molto precisa e pedante.

25. ◊◊◊ Partendo dalla definizione fornita nella sezione 1.10.1 si elabori un algoritmo, descritto in linguaggio naturale, atto alla valutazione se un determinato anno è bisestile o meno. La definizione deve terner conto del fatto che gli anni divisibili per 4000 non sono bisestili.

26. ◊◊◊ Si descriva in linguaggio naturale un algoritmo che replichi la struttura illustrata in fig. 1.17.

27. ◊◊◊ Si descriva in linguaggio naturale un algoritmo che replichi la struttura illustrata in fig. 1.18, enfatizzando le differenze con la descrizione fornita nell'esercizio precedente.

28. ◊◊◊ Si riformuli il problema delle 12 monete, illustrato nell'omonimo esercizio guidato, utilizzando solo 4 monete con 2 pesate a disposizione.

29. ◊◊♦ Si descriva, utilizzando un grado di dettaglio elevato a piacere, l'azione "apro la porta".

30. ◊◊♦ Si descriva in linguaggo naturale un algoritmo, alternativo a quello indicato nell'esercizio guidato "Calcolo dell'area di un poligono irregolare", che valuti se due segmenti di retta posti su un piano cartesiano si intersecano oppure no.

Diagrammi di flusso

Si documentino sotto forma di diagrammi di flusso gli esercizi sulla pseudocodifica (vedi pag. 89) dal numero 1 al numero 10.

1. ◊◊◊ Si elenchino e si disegnino i principali simboli utilizzati nei diagrammi di flusso.

2. ◊◊◊ Si illustrino le differenze fra il blocco di Elaborazione e quello di In/Out.

3. ◊◊◊ Si spieghi perchè la struttura di fig. 1.17 va evitata.

4. ◊◊◊ Si tracci il diagramma di flusso dell'algoritmo di Euclide.

5. ◊◊◊ Si tracci il diagramma di flusso dell'algoritmo di divisione intera. Si faccia uso, se necessario, delle *subroutines*.

6. ◊◊◊ Dati due numeri x e y, con $x \in N, x \mid x > 0$ e $y \in N, y \mid y \geq 0$ si tracci il diagramma di flusso dell'algoritmo di elevamento a potenza x^y mediante le sole 4 operazioni aritmetiche.

7. ◊◊◊ Data una successione di n numeri naturali a_1, a_2, .. a_n, si tracci il diagramma di flusso che trova il massimo di detti numeri.

8. ◊◊◊ Data una successione di n numeri naturali a_1, a_2, .. a_n, si tracci il diagramma di flusso che calcola la media aritmetica di detti numeri.

9. ◊◊◊ Data una successione di n numeri naturali a_1, a_2, .. a_n, si tracci il diagramma di flusso che somma i soli numeri dispari.

10. ◊◊◊ Data una successione di n numeri naturali a_1, a_2, .. a_n, si tracci il diagramma di flusso che somma i soli numeri con indice dispari.

11. ◊◊◊ Si tracci il diagramma di flusso dell'algoritmo atto a calcolare la somma dei numeri naturali pari (ossia 2+4+6 ..) compresi fra 0 e n. Si ponga attenzione al fatto che n può essere sia pari che dispari.

12. ◊◊◊ Si tracci il diagramma di flusso dell'algoritmo atto a calcolare la somma dei numeri naturali divisibili per 3 (ossia 3+6+9 ..) compresi fra 0 e n. Si ponga attenzione al fatto che n può non essere divisibile per 3.

13. ◊◊◊ Si consideri l'esercizio illustrato a pag. **??**. Si tracci il diagramma di flusso che permette il calcolo del numero di passi necessari per eseguire l'algoritmo. Per $n = 17$ il numero di passi rappresentati in tab. **??** è 24.

14. ◊◊◊ Un numero di due cifre è tale che la somma delle cifre di cui è composto vale 10 e la differenza fra la cifra delle decine e quella delle unità $(C_d - C_u)$ vale n, con n pari e $n \leq 10$. Si tracci un diagramma di flusso che permetta di calcolare detto numero in funzione di n.

15. ◊◊◊ La frazione generatrice di un numero decimale periodico è la frazione avente per numeratore il numero dato privato della virgola e diminuito del numero formato da tutte le cifre che precedono il periodo e per denominatore il numero formato da tanti 9 quante sono le cifre del periodo seguiti da tanti zeri quante sono le cifre dell'antiperiodo. Lo studente definisca l'algoritmo, in forma di diagramma di flusso, che permette il calcolo della frazione generatrice.

16. ◊◊◊ Si tracci il diagramma di flusso di un algoritmo atto a valutare se un determinato numero $n \in \mathbb{N}, n | 2 \leq n \leq 1000$ è primo o meno.

17. ◊◊◊ Si tracci il diagramma di flusso di un algoritmo atto a valutare se un determinato numero $n \in \mathbb{N}, n | 2 \leq n \leq 1000$ è un quadrato perfetto o meno. Si eviti, a tal fine di utilizzare funzioni predefinite che calcolino la radice quadrata in maniera approssimata o esatta.

18. ◊◊◊ Si elabori il diagramma di flusso di un algoritmo che permette di calcolare la somma di due vettori. Si identifichino detti vettori mediante un angolo compreso fra 0 e 2π radianti e mediante il relativo modulo. In tal modo il verso del vettore sarà implicito (si immagini la coda al centro del cerchio gognometrico). Se lo studente non possiede ancora le necessarie nozioni di trigonometria si utilizzino solamente i seguenti angoli:

$\pi/6$	$\pi/4$	$\pi/3$	$\pi/2$	$2\pi/3$
$3\pi/4$	$5\pi/6$	π	$7\pi/6$	$\pi/6$
$5\pi/4$	$4\pi/3$	$3\pi/2$	$5\pi/3$	$7\pi/4$
$11\pi/6$	2π			

19. ◊◊♦ Dato un numero naturale n, si tracci il diagramma di flusso dell'algoritmo che calcola la somma delle singole cifre del numero n. Se, ad esempio, $n = 1234$, il risultato deve essere 10.

20. ◊◊♦ Si definisca il diagramma di flusso del seguente algoritmo, che **dovrebbe** permettere di aggirare un ostacolo posto sul percorso di un robot:

    (a) procedere verso nord finché non si incontra un ostacolo;
    (b) girare a sinistra (di 90°) finché l'ostacolo non sia posto alla propria destra;
    (c) girare intorno all'ostacolo (rotazioni multiple di 90°) finché non sarà possibile riprendere verso nord;
    (d) tornare al punto a).

    L'algoritmo dato permette di aggirare qualsiasi ostacolo? Si giustifichi la risposta.

21. ◊◊♦ Si definisca il diagramma di flusso del seguente algoritmo, che permette di aggirare un ostacolo posto sul percorso di un robot (Algoritmo di Pledge: tratto da ABELSON-DISESSA [**?**]):

    (a) procedere verso nord finché non si incontra un ostacolo;
    (b) girare a sinistra (di 90°) finché l'ostacolo non sia posto alla propria destra;
    (c) girare intorno all'ostacolo (rotazioni multiple di 90°), tenendolo sempre sulla destra, finché la rotazione totale (compresa la rotazione di cui al punto c) non sarà pari a 0°;
    (d) tornare al punto a).

22. ◊◊♦ Si pongano a confronto i due algoritmi precedenti e si rilevino gli eventuali punti di forza di ciascuno. Si tracci il diagramma di flusso che descriva l'algoritmo di scelta dell'uno o dell'altro criterio di aggiramento dell'ostacolo.

23. ◊◊♦ Si chiede la definizione dell'algoritmo, documentato in forma di diagramma di flusso, che permette il gioco del MU. Esso consiste, data la successione di lettere (stringa) di partenza MI, nell'arrivare, attraverso la rigida applicazione di 4 regole, alla stringa di arrivo MU. Il gioco è spiegato in HOFSTADTER [**?**] e le regole sono le seguenti:

    (a) REGOLA I. Se si possiede una stringa che termina con una I, si può aggiungere una U alla fine. Esempio: da MI si può passare a MIU; da MIIUIUUI si può passare a MIIUIUUIU;
    (b) REGOLA II. Si abbia Mx (dove x è una qualsiasi successione di lettere permesse o anche una singola lettera). Allora si può ottenere Mxx. Esempio: da MUM si può passare a MUMUM; da MI si può passare a MII;
    (c) REGOLA III. Se in una stringa c'è la successione III, si può costruire una nuova stringa ponendo U al posto di III. La regola non è utilizzabile a rovescio: non si può sostituire una U con III. Esempio: da UMIIIMU si può passare a UMUMU; da MIIII si può passare a MIU o MUI;
    (d) REGOLA IV. Se all'interno di una delle stringhe c'è UU, si può eliminare detta successione. Esempio: da MUUU si può passare a MU; da MUUUII si può passare a MUII.

24. ◊◊♦ Si tracci il diagramma di flusso del problema delle 12 monete illustrato nell'omonimo esercizio guidato.

25. ◊◊♦ Si tracci il diagramma di flusso del calcolo dell'area di un poligono irregolare di n lati mediante la formula dell'area di Gauss, avente la seguente struttura:

$$A_p = \frac{1}{2} \left| \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i) \right| \tag{1.63}$$

    dove si intende che $x_{n+1} = x_1$ e $y_{n+1} = y_1$, essendo x_i e y_i le coordinate x e y del vertice i.

26. ◊◊♦ I babilonesi, 2000 anni prima dell'era cristiana, erano già in grado di calcolare la radice quadrata approssimata (cfr. BOYER [**?**]). Si supponga di voler calcolare $x = \sqrt{a}$. Allora:

    (a) sia a_1 la radice approssimata, ad esempio per difetto, di a;
    (b) si calcola $b_1 = \frac{a}{a_1}$;
    (c) si calcola la media aritmetica fra a_1 e b_1: $a_2 = \frac{1}{2}(a_1 + b_1)$;
    (d) si calcola $b_2 = \frac{a}{a_2}$;
    (e) si torna al punto c) e si termina l'algoritmo dopo un opportuno numero di iterazioni in relazione alla precisione voluta. Il risultato è contenuto in a_n.

    Si tracci il diagramma di flusso dell'algoritmo e si spieghi perché funziona.

27. ♢♢♦ Si tracci il diagramma di flusso di un algoritmo che converta un numero romano in notazione araba decimale.

28. ♢♢♦ Si tracci il diagramma di flusso di un algoritmo che converta un numero decimale in numero romano.

29. ♢♢♦ Si immagini un software applicativo per PC che, data un'ora immessa nella forma *hh:mm*, visualizzi detta ora mediante un orologio analogico. Si tracci il diagramma di flusso dell'algoritmo che a) calcola l'angolo (rispetto alle ore 12) formato dalla lancetta dei minuti e b) calcola l'angolo (rispetto alle ore 12) formato dalla lancetta delle ore.

30. ♢♦♦ Sia dato il triangolo rettangolo ABC di fig. 1.73 avente i seguenti lati: $\overline{AC} = 3$, $\overline{CB} = 4$ e $\overline{BA} = 5$.

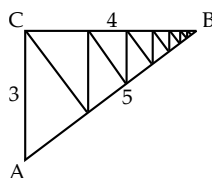



Figura 1.73: Suddivisione triangolo rettangolo

Si tracci il diagramma di flusso dell'algoritmo che calcoli l'area di ciascuno degli n triangoli rettangoli nei quali esso è stato suddiviso. Si verifichi la correttezza dei calcoli sommando l'area dei triangoli e confrontandola con l'area di ABC. Qual è la relazione fra un triangolo qualsiasi e quello immediatamente più piccolo? Qual è la relazione fra l'ultimo triangolo (quello con vertice B) e quello immediatamente precedente?

Pseudocodifica

Si documentino sotto forma di pseudocodifica gli esercizi sui diagrammi di flusso dal numero 4 al numero 30.

1. ♢♢♢ Un modo alternativo di enunciare l'algoritmo di Euclide per il calcolo del MCD è il seguente:

   (a) siano dati due numeri interi positivi n e m;
   (b) se n e m sono uguali, l'algoritmo termina e il MCD è dato dai due numeri;
   (c) altrimenti si deve sotrarre il numero minore da quello maggiore e scartare il maggiore;
   (d) si deve ricominciare il processo dal punto b), utilizzando il minore dei due numeri e la differenza calcolata al passo precedente.

   Si esprima l'algoritmo sotto forma di pseudocodifica.

2. ♢♢♢ Si esprima in pseudocodifica la versione "informatica" dell'algoritmo esteso di Euclide.

3. ♢♢♢ Si esprima in pseudocodifica la versione "matematica" dell'algoritmo esteso di Euclide.

4. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.63.
5. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.64.
6. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.65.
7. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.67.
8. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.68.
9. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.69.
10. ◊◊◊ Si traduca in pseudocodifica il diagramma di flusso di fig. 1.70.

Parte II

I FONDAMENTI

Capitolo 2

I tipi di dato

Sui banchi di scuola ci è stato insegnato che la matematica classifica le variabili secondo le loro caratteristiche. Abbiamo così imparato a distinguere gli insiemi dei numeri naturali, relativi, razionali, reali e complessi, e le relative operazioni matematiche ad essi associate e permesse.

Abbiamo anche imparato che alcuni insiemi numerici sono maggiormente adatti per rappresentare determinate situazioni reali, mentre altri insiemi lo sono di meno.

Se dovessimo misurare una distanza, molto probabilmente eviteremmo di indicarla con la notazione $(3 + j4)$ cm. Con ogni probabilità ripiegheremmo su un più tradizionale 5 cm, anche se le due notazioni indicano la stessa identica misura. Analogamente, se dovessimo identificare un vettore, la notazione complessa sarebbe sicuramente più indicata di quella, ad esempio, reale. Con la prima, infatti, potremmo esprimere non solo il modulo del vettore (ossia la "distanza" che esso copre) ma, implicitamente, anche la sua direzione ed il suo verso. Utilizzando la notazione reale, invece, si esprimerebbe solamente l'informazione legata al modulo, lasciando inespressa la direzione ed il verso.

Se si sposta l'attenzione dalla matematica all'informatica, l'importanza che assumono le caratteristiche delle variabili sono addirittura accentuate. In tale ambito le diverse variabili sono raggruppate in *tipi* anziché insiemi. Non solo: oltre ai tipi predefiniti, che in ambito informatico sono identificati come *tipi primitivi*, è possibile *dichiarare* nuovi tipi, a seconda delle proprie esigenze.

E proprio i diversi *tipi primitivi* di variabili e la loro *dichiarazione* e/o *definizione* rappresentano l'argomento delle prossime sezioni.

2.1 La dichiarazione e la definizione

Il linguaggio C, analogamente a quanto avviene anche per altri linguaggi, prevede che tutti gli oggetti debbano essere *dichiarati* ed altri debbano essere *definiti*. Si tratta quindi di chiarire[1] tali concetti:

Definizione 1 (Dichiarazione)**.**
Dichiarare una variabile significa eseguire due azioni implicite, ossia trasparenti al programmatore:

- ***associare*** *ad una variabile un identificatore;*
- ***stabilire*** *le caratteristiche della variabile (insieme di appartenenza, operatori permessi, ecc.).*

Analogamente si può provare a chiarire anche il concetto di *definizione* di variabile. Purtroppo, le prossime righe ospiteranno un orribile gioco di parole che, però, non si sa come aggirare, per cui lo si propone sfacciatamente:

Definizione 2 (Definizione)**.**
Definire una variabile significa eseguire tre azioni implicite, ossia trasparenti al programmatore:

- ***associare*** *ad una variabile un identificatore;*
- ***stabilire*** *le caratteristiche della variabile (insieme di appartenenza, operatori permessi, ecc.);*
- ***allocare*** *adeguato spazio in memoria per la variabile.*

Come si vede la dichiarazione e la definizione sono piuttosto simili: la prima non alloca spazio in memoria, la seconda invece sì. Come si possano distinguere i due concetti verrà chiarito man mano che sarà utile e necessario farlo. Momentaneamente è sufficiente impadronirsi del secondo concetto, quello di definizione. Un esempio di definizione di variabile in linguaggio C potrebbe essere quello evidenziato in fig. 2.1, che si presta già alle prime riflessioni.

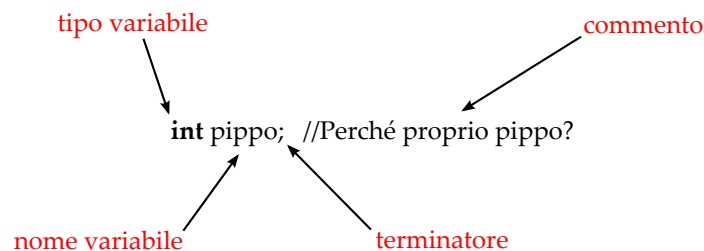



Figura 2.1: Esempio di definizione di variabile

Il primo *identificatore*[2] è riferito al **tipo** della variabile. Nel presente caso l'identificatore `int` è predefinito e rappresenta una *parola chiave*, ossia un identificatore che non può essere utilizzato dal programmatore con diverso significato da quello predefinito.

[1]Moltissimi sono i termini ed i correlati concetti che andrebbero seriamente definiti parlando di linguaggio C. Si decide, didatticamente, di definire formalmente solo i concetti più importanti, fornendo semplicemente una spiegazione discorsiva ed informale nei casi ritenuti meno importanti.

[2]Il concetto di identificatore e le regole ad esso associate saranno oggetto della prossima sezione.

Il secondo indentificatore, separato dal primo mediante uno spazio (che assume, appunto, il ruolo di *separatore*), è riferito alla variabile. Esso rappresenta, quindi, esattamente come in matematica, il *nome* della variabile. Ogni qualvolta ci si deve riferire ad essa, si utilizzerà il suo nome.

La definizione si conclude sintatticamente con il *terminatore*, ossia il punto e virgola (**;**). Esso ha il compito di indicare al compilatore dove l'*espressione* finisce. Dal punto di vista strettamente semantico lo studente compie, frequentemente, un classico errore, attribuendo al terminatore il ruolo di indicatore della fine dell'*istruzione*. Ciò è fortemente riduttivo: il linguaggio C è un linguaggio basato su *espressioni*, non su *istruzioni* e il terminatore ha il compito di indicare la fine dell'espressione, non solo dell'istruzione.

L'ultimo componente, facoltativo, della dichiarazione evidenziato in fig. 2.1 è il commento. Esso ha il compito di sciogliere eventuali ambiguità legate all'espressione a cui si riferisce. Uno dei modi per identificarne l'inizio è dato dal metasimbolo **//**. In tal caso il commento termina a fine riga. Su tale argomento si ritornerà diffusamente nella sezione 3.6. Piuttosto si vuole molto brevemente entrare nel merito del commento. Perché "pippo"? Negli esempi si useranno frequentemente degli identificatori piuttosto improbabili o fantasiosi. La scelta è voluta, per decontestualizzare l'identificatore. Si spera che ciò scoraggi lo studente ad "inventarsi" le regole da solo, magari a causa di una improvvida associazione di idee effettuata con il nome utilizzato.

Ad esempio, se al posto di **pippo** si sostituisse l'identificatore **numero**, lo studente potrebbe erroneamente ipotizzare che al tipo `int` si possano associare solamente variabili che rappresentano numeri, mentre ciò è riduttivo e anche impreciso. L'uso di nomi improbabili dovrebbe evitare l'insorgere di tali ambiguità.

Da quanto fin qui detto, dovrebbe essere evidente che l'*associazione* alla variabile viene effettuato attraverso il suo **identificatore**. Ogni qualvolta, quindi, dovremo riferirci ad una determinata variabile, lo faremo attraverso il suo identificatore, avente il compito di *associare* la variabile al suo nome.

Le caratteristiche della variabile sono stabilite attraverso il suo **tipo**. Esso definisce in maniera inequivocabile:

- l'**insieme di appartenza** della variabile, ossia i valori consentiti dall'insieme. Una variabile di tipo `int` (intero, associabile in matematica all'insieme dei numeri relativi), ad esempio, può assumere solamente valori interi dotati di segno;
- le **operazioni** associate alla variabile stessa e quindi, di conseguenza, anche gli operatori permessi nelle espressioni utilizzanti detta variabile. Ad esempio, fra variabili di tipo intero non è permessa la divisione decimale, ma solo la divisione intera con resto;
- lo **spazio** necessario in memoria per "ospitare" la variabile. Ci si deve sempre ricordare che, contrariamente alla matematica, l'informatica tratta sempre valori approssimati ed espressi mediante un numero finito di cifre e che queste cifre necessitano di uno spazio opportuno per essere memorizzate. Ad esempio, le variabili di tipo `int`, a seconda del sistema operativo, dell'ambiente di sviluppo e degli *specificatori di tipo*[3] utilizzati, occupano 32 oppure 64 bit. In altri casi, come in certi ambienti di sviluppo per microcontrollori, si può arrivare anche a soli 16 bit.

[3] Argomento trattato nella sezione 2.3.5.

Allocare spazio in memoria per una determinata variabile significa *ricercare* e *riservare* adeguato spazio in memoria per memorizzare la variabile data. Dette azioni sono eseguite in maniera trasparente al programmatore, che non ha la necessità di preoccuparsi di tali problemi.

Esempi corretti ed errati di definizione di variabile sono i seguenti:

Corretto	Errato
int pippo;	int pippo
int pippo, pluto;	int pippo; pluto;
int pippo ;	pippo int;

Tabella 2.1: Esempi corretti ed errati di definizione di variabile

2.2 L'identificatore

Quale sia il compito che l'identificatore è chiamato ad assolvere è stato illustrato nella sezione precedente. Nella presente si tratteranno le regole che ne vincolano la definizione. Sostanzialmente si devono osservare **tre regole** per definire correttamente un identificatore:

1. un identificatore deve sempre iniziare con lettera maiuscola o minuscola dell'alfabeto inglese. Si tenga presente che il carattere *underscore* (_) è interpretato come lettera corretta;
2. dal secondo carattere in poi possono essere (solo) usati gli *underscore*, le lettere maiuscole e minuscole dell'alfabeto inglese e le cifre da 0 a 9;
3. la lunghezza è limitata a 31 caratteri per le variabili *interne* e 6 caratteri per le variabili *esterne*[4].

Queste semplici regole vanno comunque commentate.

Perché si parla di alfabeto inglese? Perché vanno evitati i caratteri accentati. Indicando l'alfabeto inglese si individua un alfabeto che contiene praticamente tutte le lettere utizzate negli alfabeti occidentali[5] senza che siano coinvolti gli accenti tipici, ad esempio, della lingua italiana, francese, spagnola, ecc. Si noti, inoltre, che il linguaggio C è *case sensitive*, ovvero distingue fra l'identificatore **PipPo** e **piPpo**.

Una seconda riflessione doverosa è dedicata all'uso dell'*underscore* in prima posizione. Pur essendo permessa, se ne sconsiglia l'uso, perché il *Linker* usa anteporre all'identificatore utilizzato dal programmatore proprio l'*underscore*, per cui, se il programmatore dovesse definire gli identificatori **pippo** e **_pippo** potrebbe essere generato un errore da parte del *Linker* di non facile localizzazione.

L'*underscore* è, invece, utilissimo per separare le singole parole connesse insieme a formare un unico identificatore come, ad esempio, **numero_primo**.

[4]Sul concetto di variabili interne ed esterne si ritornerà nei prossimi capitoli.

[5]L'unica eccezione, tutto sommato irrilevante, che viene alla mente è la Ø danese.

L'ultima riflessione è riservata alla lunghezza degli identificatori. Il linguaggio ANSI C11 *distingue* identificatori interni di massimo 31 caratteri e indentificatori esterni di massimo 6 caratteri. Ciò non significa che non possono essere utilizzati identificatori con più di 31 caratteri, ma che solamente i primi 31 vengono utilizzati per discernere una variabile dall'altra.

Comunque la maggior parte dei *dialetti*[6] ha superato questo problema.

2.3 I tipi primitivi

Il linguaggio C possiede solamente 4 distinti tipi primitivi:

- `char`, formato da 8 bit e usato prevalentemente per rappresentare caratteri;
- `int`, formato da 16, 32 o 64 bit e usato prevalentemente per rappresentare numeri interi;
- `float`, formato da 32 bit e usato prevalentemente per rappresentare numeri decimali;
- `double`, formato da 64 bit e usato prevalentemente per rappresentare numeri decimali con maggior precisione.

I tipi `char` e `int`, a loro volta, possono essere espressi con segno e senza segno, ovvero *signed* e *unsigned*. Il programmatore può quindi scegliere quale rappresentazione è maggiormente adatta ad un determinato contesto. I tipi `float` e `double` sono, invece, sempre espressi con segno e non è possibile esprimerli solamente mediante modulo.

Inoltre, è possibile effettuare un'altra distinzione fra i tipi primitivi: i tipi `char` e `int` rappresentano sempre **valori interi** che possono essere espressi o in modulo (*unsigned*) o in complemento a due (*signed*); i tipi `float` e `double` rappresentano invece dei **valori decimali** e sono espressi in virgola mobile (*floating point*).

Questa ulteriore distinzione introduce le prossime due sezioni. Il primo argomento che verrà trattato è il complemento in base due, che sarà, però, preceduto da un altro argomento, propedeutico al complemento a due: il complemento alla base.

2.3.1 Il complemento alla base

Un concetto che tornerà molto utile è quello di **complemento alla base**. Esso permette di aggirare le difficoltà legate all'operazione di sottrazione aritmetica, come si vedrà fra breve.

[6]Per "dialetto" si intende un linguaggio che presenta piccole differenze con il riferimento standard (ANSI C, C89, C99, IEC/ISO 9899, IEC 14882, ecc.).

Il complemento a 9 di un numero in base 10 si ottiene sottraendo da 9 ciascuna cifra del numero dato, come illustrato nell'esempio seguente:

$$\begin{aligned} &999- \\ &345 = \\ &654 \end{aligned} \tag{2.1}$$

quindi il complemento a 9 di 345 è 654.

Il **complemento a 10** di un numero in base 10 è il complemento a 9 più 1. Ciò appare evidente se nell'espressione aritmetica 2.1 si somma 1 ad ambo i membri:

$$\begin{aligned} 999{+}1- \\ 345 = \\ 654{+}1 \end{aligned} \tag{2.2}$$

Quindi il complemento a 10 di 345 è 655 (654 + 1).

E' ora possibile introdurre la **somma in complemento a 10**. Si supponga di voler effettuare l'operazione aritmetica 7 – 4, ma di non conoscere l'operatore di sottrazione e di non sapere come si effettuano le sottrazioni aritmetiche.

E' possibile aggirare il problema effettuando una somma in complemento a 10. Essa si ottiene eseguendo il complemento a 10 del sottraendo (4), sommando il termine così ottenuto al minuendo (7) e trascurando la cifra più significativa del risultato.

Di seguito si fornisce un esempio numerico di quanto appena esposto, confrontando una classica sottrazione con la corrispondente somma in complemento a 10:

7 -		7 +
4 =	⇒	6 =
3		~~1~~3

Figura 2.2: Somma in complemento a 10

Invece di sottrarre 4 da 7 si può sommare a 7 il complemento a 10 di 4, cioè 6. Il risultato di tale operazione è 13. Trascurando la cifra più significativa del risultato (1) si ottiene il risultato della sottrazione (3).

Un ulteriore esempio è il seguente:

123 -		123 +
45 =	⇒	955 =
78		~~1~~078

Figura 2.3: Ulteriore esempio di somma in complemento a 10

Il complemento a 10 di 45 equivale al suo complemento a 9 (complemento a 9 di ciascuna cifra) più 1, quindi 955. Sommando 123 a 955 si ottiene 1078. Ignorando la cifra più significativa (1) si ottiene il risultato della sottrazione originale (78).

Naturalmente l'esempio formulato in base 10 vale anche per le altre basi. Si veda a tal proposito la sezione successiva.

2.3.2 Il complemento a due

Tutti i microprocessori ed i microcontrollori non eseguono mai sottrazioni aritmetiche, ma sempre somme in complemento a due.[7]

Tale aspetto dell'aritmetica binaria relativa alle macchine microprogrammate non è eludibile, per cui conviene dedicare qualche riga aggiuntiva per sottolineare l'importanza dell'argomento e permettere allo studente di familiarizzare con esso. Si veda a tal proposito il seguente esercizio.

Esercizio - ◊◊◊ Somma in complemento a 2

Si supponga di voler eseguire la sottrazione di due numeri in base 2 indicati nella espressione 2.3, ma di non voler utilizzare (o non poter utilizzare) l'operatore di sottrazione.

$$01010 - 00011 \tag{2.3}$$

In tal caso è possibile aggirare il problema, utilizzando la **somma in complemento a 2**:

Soluzione

La sottrazione (2.3) in base 2 equivale alla sottrazione decimale $10 - 3$ e deve quindi dare come risultato 7, ossia 0111_2.

La prima cosa da effettuare è il complemento a 2 del sottrattore. Ciò si esegue calcolando il complemento alla base-1 (ossia il complemento a 1) e sommando 1 al risultato.

Quindi, il calcolo del complemento a 1 del numero binario 0011, diventa il seguente:

$$\begin{aligned} &11111- \\ &00011 = \\ &11100 \end{aligned} \tag{2.4}$$

Si noti che 11100 è null'altro che la negazione delle singole cifre del numero originale 00011, dove ciascuno 0 è diventato 1, e ciascun 1 è diventato 0.

Aggiungendo 1 al complemento a 1 si ottiene il complemento a 2 del numero. Quindi:

$$\begin{aligned} 11100&+ \\ 1&= \\ 11101& \end{aligned} \tag{2.5}$$

[7] Il problema nasce da un'esigenza reale: le unità di calcolo (ALU) sono velocissime ad effettuare le somme ed i complementi ad 1, mentre non sono progettate per eseguire le sottrazioni. La somma in complemento a due permette di mantenere alte le velocità di elaborazione di espressioni aritmetiche anche in presenza di sottrazioni.

che rappresenta proprio il complemento a 2 di 00011.

Ora è possibile eseguire la somma in complemento a 2 anziché eseguire la sottrazione indicata in (2.3):

$$\begin{aligned} 01010+ \\ 11101 = \\ 100111 \end{aligned} \tag{2.6}$$

Siccome la cifra più significativa va ignorata, il risultato finale è 00111_2 che equivale proprio a 7_{10}, che è il risultato cercato.

Dato che il concetto di somma in complemento alla base non dipende dalla base utilizzata, si propone un esercizio di somma in complemento a 16. Lo scopo non è quello di aggiungere un ennesimo esercizio con base non ancora usata, ma far emergere un'apparente contraddizione e fornire poi l'opportuna spiegazione.

Esercizio - ◊◊♦ Somma in complemento a 16

Si supponga di voler eseguire la seguente sottrazione di due numeri in base 16, utilizzando la somma in complemento a 16:

$$6AC5 - 49FD \tag{2.7}$$

Soluzione

Il calcolo del complemento a 15 di $49FD_{16}$ si esegue complementando alla base-1 ogni singola cifra:

$$\begin{aligned} FFFF- \\ 49FD = \\ B602 \end{aligned} \tag{2.8}$$

Analogamente a quanto fatto nell'esercizio precedente, a detto risultato si deve ora aggiungere 1 per ottenere il complemento a 16:

$$\begin{aligned} B602+ \\ 1 = \\ B603 \end{aligned} \tag{2.9}$$

Ora è possibile eseguire la somma in complemento a 16:

$$\begin{aligned} 6AC5+ \\ B603 = \\ 120C8 \end{aligned} \tag{2.10}$$

Ignorando la cifra più significativa si ottiene il risultato della sottrazione (2.7): $20C8$.

Lo studente attento potrebbe obiettare che le operazioni (2.4) e (2.8) sono a tutti gli effetti delle sottrazioni, per cui non ha senso introdurre la somma in complemento alla base per evitare le sottrazioni se poi, per effettuare il complemento alla base, se ne deve eseguire una.

La realtà è leggermente diversa e dal punto di vista pratico non viene effettivamente eseguita alcuna sottrazione.

La base 16 è usata solamente come strumento di rappresentazione di un numero che nella sostanza è binario. Nessun calcolatore esegue, in realtà, operazioni aritmetiche in detta base, ma sempre e solamente in base 2. In aritmetica binaria, però, il complemento a 1 implica semplicemente la negazione di tutte le cifre binarie e non una sottrazione vera e propria.

Nei presenti esercizi si continua a parlare di sottrazione per poter generalizzare il problema e per mantenere una correlazione con il nostro sistema decimale. Ma nella realtà (cioè all'interno dell'unità di calcolo di un PC) non viene mai eseguita alcuna sottrazione.

2.3.3 La rappresentazione in virgola mobile

La rappresentazione classica dei numeri, secondo la quale il numero viene indicato scrivendo diligentemente ciascuna cifra, qualsiasi ne sia il numero, non è sempre la migliore soluzione. Un numero molto grande oppure molto piccolo, può risultare di non immediata lettura, come nei due esempi seguenti:

$$1000000000 \tag{2.11}$$

$$0.000000001 \tag{2.12}$$

Istintivamente, il lettore pignolo avrà la tendenza a contare il numero di zeri al fine di non commettere errori di lettura del numero.

La rappresentazione del numero risulta essere più efficace se si usa la notazione scientifica[8], come nei rispettivi due esempi:

$$1 \cdot 10^{9} \tag{2.13}$$

$$1 \cdot 10^{-9} \tag{2.14}$$

Quest'ultima rappresentazione risulta essere efficiente anche in termini di memorizzazione umana, oltre che di lettura.

Anche la memorizzazione dei numeri nei calcolatori soffre degli stessi problemi. Mentre può essere efficiente memorizzare il numero 123456789 nello stesso modo in cui lo si rappresenta, sorge il legittimo dubbio che non lo sia per il numero 1000000000.

Nel tentativo di risolvere i suddetti problemi, l'*Institute of Electrical and Electronics Engineers* (IEEE) ha elaborato nel 1985 uno standard denominato IEEE 754 per la rappresentazione dei numeri in virgola mobile (*floating point*).

---

[8]Vi sono altre notazioni molto usate in ambito scientifico e tecnico, come , ad esempio, la *notazione ingegneristica*, utile soprattutto per la rappresentazione di grandezze fisiche, che però non verrà trattata nelle presenti pagine.

Prima, però, di illustrare tale standard è opportuno fornire qualche brevissima nozione teorica.

Un numero reale in base b può essere espresso in notazione esponenziale nella seguente forma

$$nr = (-1)^s \cdot f \cdot b^e \quad con \quad s = \{0, 1\}, \quad e \in \mathbb{Z} \tag{2.15}$$

dove s condiziona il *segno*, b è la *base* di rappresentazione, e è un *esponente* intero e f è la *mantissa*, scritta in modo tale che

$$1/b \leq f < 1 \tag{2.16}$$

dove quest'ultimo limite imposto ad f è detto *normalizzazione*.

Quanto detto permette di affrontare il primo esempio.

Esercizio - ♢♢♦ Conversione virgola mobile-decimale

Si vuole convertire il numero 123.45 in virgola mobile a base 10.

Soluzione

E' sufficiente attenersi alle due relazioni di riferimento 2.15 e 2.16. Quindi

1. essendo la base 10, si ha $b = 10$;
2. essendo il numero da rappresentare positivo, si assegna $s = 0$;
3. normalizzare 123.45 secondo la relazione 2.16 significa semplicemente dividerlo per 1000, per cui $f = 0.12345$;
4. la normalizzazione implica $e = 3$;
5. applicando la relazione 2.15 si ottiene: $nr = (-1)^0 \cdot 0.12345 \cdot 10^3$

Convertire, quindi, un numero reale in base 10 in numero *floating point* della stessa base è piuttosto semplice.

Le prossime pagine affronteranno in maggior dettaglio la notazione in virgola mobile ed in particolare lo standard IEEE 754, nonché le notazioni adottate dalla Microchip per la rappresentazione in virgola mobile a 24 e 32 bit, senza tuttavia addentrarci negli algoritmi relativi alle operazioni aritmetiche vere e proprie, ma limitandosi alle conversioni di rappresentazione.

Benché l'argomento non sia del tutto elementare, si ritiene ugualmente molto utile uno studio non eccessivamente approfondito o specialistico di detto standard, in totale armonia con il pensiero espresso da uno dei più grandi divulgatori e studiosi della *Computer Science*:

Every well-rounded programmer ought to have a knowledge of what goes on during the elementary steps of floating-point arithmetic. This subject is not at all as triavial as most people think, and it involves a surprising amount of interesting information.

Donald E. Knuth - *The Art of Computer Programming, Vol 2*

2.3.3.1 I numeri di macchina

Un primo problema da affrontare è dato dal fatto che ciascun numero reale, nei sistemi a microprocessore e microcontrollore, è memorizzato in una porzione di memoria finita ed uguale per tutti i numeri reali. Pertanto il numero π ed il numero 1.0 utilizzano la stessa quantità di memoria per la rispettiva memorizzazione e successiva rappresentazione.

Ciò significa che alcuni numeri devono essere memorizzati in forma troncata ed approssimata, non potendo assegnare loro una quantità infinita di memoria.

Si noti, inoltre, che il problema, in informatica, non si presenta solamente per i numeri trascendenti ma anche per molti numeri apparentemente innocui, se rappresentati in base 10, che assumono rappresentazioni molto più complesse o addirittura periodiche, come evidenziato nel seguente esercizio.

Esercizio - ◊◊◊ Perdita d'informazione

Si supponga di voler rappresentare il numero 4/5 (ovvero 0.8) in base 2.

Soluzione

Nella forma polinomiale, la parte intera del numero si esprime mediante somma di potenze a esponente positivo e la parte decimale mediante somma di potenze a esponente negativo.

Siccome, nel numero 0.8, la parte intera è posta a 0, la parte decimale si esprime, quindi, nel seguente modo:

$$2^{-01} + 2^{-02} + 2^{-05} + 2^{-06} + 2^{-09} + 2^{-10} + 2^{-13} + 2^{-14} + 2^{-17} + 2^{-18} +$$
$$+ 2^{-21} + 2^{-22} + 2^{-25} + 2^{-26} + 2^{-29} + 2^{-30} + 2^{-33} + 2^{-34} + 2^{-37} + 2^{-38} \ldots$$

senza poter raggiungere una *rappresentazione esatta* del numero, ma solamente una rappresentazione approssimata. Dopo 50 termini (compresi i termini con coefficiente a 0) il polinomio restituisce il numero

$$0.79999999999999988818$$

che è indubbiamente un'ottima approssimazione in molti casi, ma non rappresenta esattamente il numero 0.8.

Quindi un reale avente un numero finito di cifre rappresentabili in base 10 (ad esempio 0.8_{10}), potrebbe diventare un numero periodico in base 2 ($0.\overline{1100}_2$), ed essere quindi soggetto a troncamento ed approssimazione. Qualsiasi calcolo decimale eseguito da un microprocessore soffre potenzialmente di tale problema.

Per contro, ad esempio, il numero 0.5 (rappresentabile in modo esatto mediante potenze di due) si rappresenta molto facilmente nella forma 2^{-1}. Vi sono quindi numeri che possono essere rappresentati in maniera piuttosto efficace e numeri per cui ciò non è possibile.

Alla luce di quanto detto si dà la seguente

Definizione 3 (Numeri di macchina)**.**
Sono detti numeri di macchina quei numeri reali che sono rappresentati in modo esatto in un particolare sistema di rappresentazione.

E' quindi fondamentale che lo studente distingua fra numeri reali e numeri di macchina, ricordando che questi ultimi possono, a volte, rappresentare i numeri reali in forma approssimata.

D'ora in poi, quando si parlerà di numeri di macchina si intenderà sempre una loro rappresentazione in base 2.

2.3.3.2 Lo standard IEEE 754

Lo standard IEEE 754 introduce alcune semplici varianti (come pure le notazioni usate dalla Microchip) nella teoria dei numeri in virgola mobile, aventi il compito di rendere le operazioni più semplici e veloci.

Nelle prossime pagine si parlerà della rappresentazione dei numeri in *virgola mobile con base b ed eccesso q, su p cifre*, dove

- b indica la base numerica (in particolare, $b = 2$ nelle presenti pagine);
- q rappresenta l'eccesso (che verrà spiegato tra breve).

Inoltre, la IEEE 754 prevede che, per semplificare e velocizzare i calcoli, venga sommato 1 alla mantissa (*hidden bit*), comunque definita secondo la relazione 2.16.

Con tali premesse e ponendo b e q costanti, la relazione 2.15 assume la seguente forma:

$$nm = (-1)^s \cdot (1 + f) \cdot b^{e-q} \quad con \quad s = \{0, 1\}, \quad e, q \in \mathbb{N} \tag{2.17}$$

La 2.17 è la relazione definitiva utilizzata dallo standard IEEE 754 per la rappresentazione in virgola mobile, nella quale il valore che la mantissa assume è dato dalla seguente successione numerica:

$$f = \sum_{k=0}^{n-1} a_{(k)} \cdot 2^{-(k+1)} \tag{2.18}$$

dove n rappresenta il numero di bit della mantissa e $a_{(k)}$ il $k$esimo bit della mantissa stessa, con $a_{(0)}$ bit più significativo.

Si noti, nella relazione 2.17, che la base b ha un esponente *in eccesso q* o *aumentato*. La notazione in eccesso q avviene perché l'esponente deve poter esprimere un numero dotato di segno, ma *non lo fa in complemento a due* per motivi di ottimizzazione di calcolo. Su 8 bit, il valore 0 è riservato per la rappresentazione dello 0 (si veda la 2.19), il valore 1 rappresenta il valore decimale -126, 2 corrisponde a -125, 3 a -124, ... 127 a 0, ... 255 a +127. Tale modo di convertire i numeri *signed* è detto in inglese anche *biased*, ovvero *aumentato* (la traduzione letterale sarebbe *polarizzato*).

Si noti, anche, che la 2.16 e la 2.17 non prevedono il caso, fondamentale, in cui $nm = 0$. Tale caso viene trattato come eccezione, in deroga alle espressioni 2.16 e 2.17, imponendo la **condizione di zero** di seguito formulata:

$$\text{CdZ: } e = 0 \wedge f = 0 \tag{2.19}$$

2.3.3.3 I formati dello standard IEEE 754

Lo standard IEEE 754 prevede quattro[9] diversi formati rappresentativi, di cui i due più utilizzati sono

- il formato in *singola precisione*;
- il formato in *doppia precisione*.

Il primo formato, in precisione singola, è quello uitlizzato dalle variabili di tipo `float` in linguaggio C, mentre il secondo, in precisione doppia, è quello utilizzato dalle variabili di tipo `double`. Il primo formato occupa 32 bit in memoria, ossia 4 byte, mentre il formato in doppia precisione occupa 64 bit pari a 8 byte.

Il formato relativo alla singola precisione utilizza un esponente aumentato da 8 bit e una mantissa da 23 bit. Il formato relativo alla doppia precisione, invece, utilizza un esponente aumentato da 11 bit e una mantissa da 52 bit.

Naturalmente, la principale differenza dei due formati sta nella precisione di rappresentazione dei numeri reali: il formato in singola precisione opera, in certi casi, approssimazioni superiori (ossia meno precise) rispetto agli stessi casi della doppia precisione.

I due formati hanno le caratteristiche illustrate in tab. 2.2.

Parametro	Singola Precisione		Doppia Precisione	
	Range	Bit	Range	Bit
s	0 1	1	0 1	1
e	+127 -126	8	+1023 -1022	11
f	$\sim 8.4 \cdot 10^6$	23	$\sim 4.5 \cdot 10^{15}$	52

Tabella 2.2: Caratteristiche formati in virgola mobile

Una rappresentazione grafica del significato dei vari bit è data in fig. 2.4 nella pagina successiva.

Il bit più significativo del byte 0 rappresenta il segno del numero in virgola mobile. I 7 bit meno significativi del byte 0 e il bit più significativo del byte 1 formano l'esponente in complemento a due. Il bit più significativo del byte 1 irappresenta il bit meno significativo dell'esponente. I restanti bit formano la mantissa, in cui i 7 bit meno significativi del byte 1 formano i 7 bit più significativi della mantissa e il byte 3 rappresenta gli 8 bit meno significativi della mantissa.

[9]Oltre alla singola precisione e alla doppia precisione, esistono anche i formati di singola precisione estesa e doppia precisione estesa che, però, non verranno trattati.

Per chiarire con un esempio quanto appena detto, si propone il seguente

Esercizio - ◊◊◊ Conversione binario-virgola mobile
Si supponga di voler calcolare il relativo valore in virgola mobile della seguente sequenza di 32 bit:

$$00111110\ 00100000\ 00000000\ 00000000 \tag{2.20}$$

Soluzione
E' possibile convertire il suddetto numero nella corrispondente notazione in virgola mobile nel seguente modo:

1. il numero è positivo, dato che il bit più significativo vale 0 e siccome $-1^0 = 1$, si ha che, in base alla 2.17, il numero viene moltiplicato per 1;
2. l'esponente è dato dai successivi 8 bit espressi in forma aumentata. Dato che essi assumono il valore decimale 124 (0x7C), si ha $124 - 127 = -3$;
3. Infine, la mantissa , in base alla 2.18, assume il valore $f = 2^{-2} = 0.25$;
4. quindi il valore finale diventa $x = 1 \cdot (1 + 0.25) \cdot 2^{-3} = 0.15625$.

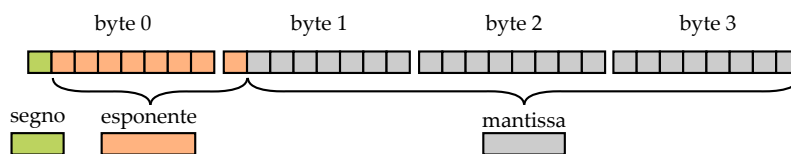



Figura 2.4: Significato dei bit in formato FP32

2.3.3.4 L'arrotondamento *round even*

Si è visto nelle sezioni precedenti che rappresentare esattamente un numero reale mediante un numero finito di bit non è sempre possibile. La rappresentazione, in molti casi, deve essere forzatamente approssimata. Nasce quindi il problema di come approssimare il numero rappresentato in virgola mobile nel migliore dei modi minimizzando gli errori sia di approssimazione che di successivo calcolo.

Le tecniche sono molteplici. Una delle più semplici, ma comunque efficace, consiste nell'arrotondamento del numero reale al numero di macchina più vicino (*rounding to nearest*), ma già negli anni '60 vennero sviluppate delle tecniche più sofisticate mediante cifre di guardia (*guard digits*). Una trattazione pregevole dell'argomento è fornita da GOLDBERG [14] indicata in bibliografia.

Innanzi tutto si deve chiarire cosa si intende per arrotondamento.

Se si deve arrotondare all'intero più vicino il numero 12.9_{10} oppure 12.1_{10} non sorgono eccessivi dubbi: nel primo caso il numero viene arrotondato a 13_{10} e nel secondo caso a 12_{10}.

Qualche fugace dubbio nasce se si deve arrotondare all'intero più vicino il reale 12.5_{10}. Esistono due scuole di pensiero a riguardo. La più nota e diffusa suddivide l'*alfabeto* decimale in due sottoinsiemii uguali: $A = \{0, 1, 2, 3, 4\}$ e $B = \{5, 6, 7, 8, 9\}$. Se la cifra da arrotondare è presente nel sottoinsieme A, allora si arrotonda *per difetto* (*round down*); se la cifra da arrotondare è presente nel sottoinsieme B, allora si arrotonda *per eccesso* (*round up*).

Esiste, però, una seconda scuola di pensiero, formalizzata da Reiser e Knuth nel 1975, che presenta alcuni fondamentali vantaggi nel calcolo in virgola mobile: siccome il valore 12.5_{10} sta esattamente a metà fra gli interi 12_{10} e 13_{10} bisognerebbe arrotondare detto valore per difetto nel 50% dei casi e per eccesso nel restante 50% dei casi.

Un semplice modo per ottenere ciò consiste nel fare in modo che il valore arrotondato sia sempre pari (*round even*). Quindi 12.5_{10} va arrotondato a 12_{10}, 13.5_{10} a 14_{10}, 14.5_{10} a 14_{10}, 15.5_{10} a 16_{10} e così via.

Per sottolineare la differenza delle due tecniche di arrotondamento ed evidenziare la bontà della seconda si propone di seguito un esercizio. Prima, però, si introducono quattro nuovi simboli per le relative operazioni aritmetiche in virgola mobile, come evidenziato nella tabella sottostante:

Operazione	Operatore Reale	Operatore FP
Somma	+	$\oplus$
Sottrazione	-	$\ominus$
Moltiplicazione	$\times$	$\otimes$
Divisione	$\div$	$\oslash$

Tabella 2.3: Opearatori reali e in virgola mobile

Quindi $3.14 + 2.71$ rappresenta un'operazione fra numeri reali e quindi esatta, mentre $3.14 \oplus 2.71$ rappresenta un'operazione in virgola mobile e quindi approssimata.

Alla luce di quanto detto, si propone il seguente

Esercizio - ◊◊♦ Confronto fra arrotondamenti

Si consideri la bontà delle due tecniche di arrotondamento precedentemente descritte (*round up*/*round down* e *round even*) nelle operazioni si somma e sottrazione in virgola mobile su base 10 con 3 cifre di precisione, utilizzando inizialmente $x = 1.00$ e $y = 0.555$.

Soluzione

Eseguendo l'operazione di somma[10] e arrotondando con metodo *round up* si ottiene

$$z = x \oplus y = 1.00 \oplus 0.555 = 1.56 \tag{2.21}$$

[10]Si considera, senza entrare nei dettagli, che l'arrotondamento abbia effettivamente luogo solo in fase di rappresentazione, ossia *dopo* che l'operazione aritmetica sia avvenuta senza arrotondamenti. Non si considerano, nelle presenti pagine, gli effettivi algoritmi di somma e sottrazione in virgola mobile.

dato che il risultato non può essere rappresentato come $z = 1.555$, ma deve essere arrotondato in 1.56 mediante *round up*.

Eseguendo la sottrazione del secondo addendo dal risultato ottenuto nella somma si dovrebbe ottenere il primo addendo, invece:

$$x = z \ominus y = 1.56 \ominus 0.555 = 1.01 \tag{2.22}$$

dato che il risultato non può essere rappresentato come $x = 1.005$, ma deve essere arrotondato in 1.01 mediante *round up*.

Insistendo nell'operazione di somma si ottiene:

$$z = x \oplus y = 1.01 \oplus 0.555 = 1.57 \tag{2.23}$$

e così via, in una pericolosa deriva che aggiunge sempre 0.01 al risultato in seguito agli arrotondamenti.

Eseguendo le stesse operazioni con arrotondamento *round even* si ottiene

$$z = x \oplus y = 1.00 \oplus 0.555 = 1.56 \tag{2.24}$$

dato che l'arrotondamento avviene in modo tale che l'ultima cifra del risultato sia pari.

Eseguendo la sottrazione del secondo addendo dal risultato ottenuto nella somma si ottiene

$$x = z \ominus y = 1.56 \ominus 0.555 = 1.00 \tag{2.25}$$

sempre in virtù del fatto che la cifra meno significativa del risultato viene arrotondata in modo che sia pari. In tal modo non si ha alcuna deriva iterando ciclicamente le due operazioni 2.24 e 2.25.

Si propongono ora ulteriori due esercizi sul tema dell'arrotondamento. Essi riguarderanno non più numeri in virgola mobile in base 10, ma numeri in virgola mobile in base 2, al fine di porre in evidenza alcuni aspetti sui quali lo studente distratto potrebbe non aver riflettuto a sufficienza. Naturalmente quanto detto a proposito della base 10 non cambia in base 2.

Esercizio - ◊◊◊ Arrotondamento *round even* per difetto

Si vuole arrotondare a 4 bit[11], mediante tecnica *round even* la seguente mantissa espressa su 5 bit: 10101.

Soluzione

La mantissa esprime la parte decimale del numero da arrotondare che, in forma polinomiale, equivale a

$$2^{-1} + 2^{-3} + 2^{-5} = 0.5 + 0.125 + 0.03125 = 0.65625_{10} \tag{2.26}$$

Si propone la situazione di arrotondamento in forma grafica, per maggior chiarezza in fig. 2.5 a fronte.

Nella parte sinistra della fig. 2.5 è rappresentata la situazione del numero prima dell'arrotondamento, mentre nella parte destra della figura è rappresentata la situazione dopo l'arrotondamento, dove il bit meno significativo del numero originale è posto in evidenza.

[11]Non si considera volutamente una mantissa di 23 bit per questioni di semplicità.

Il bit meno significativo dopo l'arrotondamento (0) ha peso $2^{-4} = 0.0625_{10}$, mentre il bit perso durante l'arrotondamento (1) avrebbe peso $2^{-5} = 0.03125_{10}$, ovvero la *metà* rispetto al bit che deve essere arrotondato.

Questa è esattamente la situazione prevista dal *round even*, in cui il numero da arrotondare (0.65625_{10}) sta esattamente a metà fra due numeri di macchina ($1010_2 = 0.625_{10}$ e $1011_2 = 0.6875_{10}$).

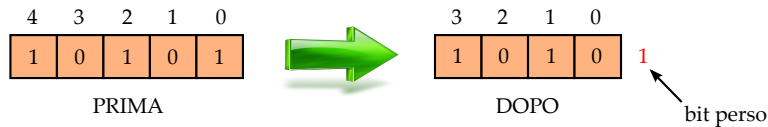



Figura 2.5: Arrotondamento *round even* (per difetto)

L'arrotondamento deve quindi avvenire per eccesso o per difetto in modo tale che il numero arrotondato sia pari. Essendo il bit meno significativo del numero arrotondato pari a 0, il numero risulta essere correttamente arrotondato e assume il valore 1010.

Il secondo esercizio sul tema dell'arrotondamento in base 2 affronta un caso leggermente più complesso. In questo caso, dopo il troncamento al quarto bit e prima dell'arrotondamento, il bit meno significativo risulta essere dispari.

Esercizio - ◊◊◊ Arrotondamento *round even* per eccesso

Si vuole arrotondare a 4 bit, mediante tecnica *round even* la seguente mantissa espressa su 5 bit: 10111

Soluzione

Anche in questo caso la mantissa esprime la parte decimale del numero da arrotondare per cui, in forma polinomiale, essa equivale a

$$2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} = 0.5 + 0.125 + 0.0625 + 0.03125 = 0.71875_{10} \quad (2.27)$$

La situazione è simile alla precedente: prima dell'arrotondamento il bit meno significativo è posto a 1, ponendo il numero da arrotondare (0.71875_{10} esattamente a metà fra due numeri di macchina, ovvero $1011_2 = 0.6875_{10}$ e $1100_2 = 0.75_{10}$.

La fig. 2.6 fornisce una illustrazione grafica della situazione prima e dopo l'arrotondamento.

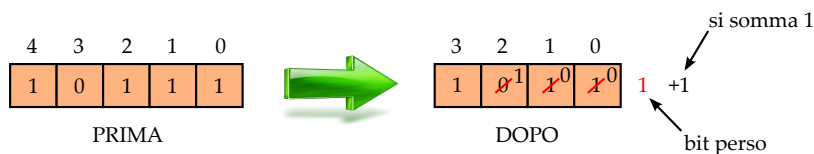



Figura 2.6: Arrotondamento *round even* (per eccesso)

In questo, però, dopo il troncamento il bit meno significativo è posto a 1, rendendo il numero dispari. Ciò significa che si deve arrotondare per eccesso la mantissa sommandole 1 (considerando la parte decimale come intera), ed ottenendo 1100, che è il corretto valore arrotondato.

La tecnica *round even* (o *rounding to the nearest*) è quella utilizzata dallo standard IEEE 754 per eseguire gli arrotondamenti.

2.3.3.5 La normalizzazione in virgola mobile

La parte più importante durante la conversione decimale-virgola mobile è sicuramente la normalizzazione, anche in virtù del fatto che ingloba l'arrotondamento.

Nei precedenti esercizi si è visto che la normalizzazione effettuata su numeri in virgola mobile su base 10 è molto semplice e anche in base due le difficoltà sono solo apparenti.

Si propone di seguito una procedura di normalizzazione in base due elaborata da Knuth (cfr. KNUTH [8]), leggermente modificata e semplificata. Al fine di generalizzare la procedura si faccia riferimento alla 2.15 e alla 2.16 a pagina 102, ponendo $b = 2$.

1. Se $|f| \geq 1$ vai al punto 4. Se $f = 0$, poni e al suo valore minimo e vai al punto 6.
2. Se $|f| \geq 1/b$ vai al punto 5.
3. Scorri a sinistra di una posizione tutti i bit di f (che equivale a moltiplicare f per due) e decrementa e. Ritorna al punto 2.
4. Scorri a destra di una posizione tutti i bit di f (che equivale a dividere f per due senza resto) e incrementa e.
5. Esegui l'arrotondamento di f. Si noti che ciò implica la possibilità che si abbia $|f| = 1$ (*rounding overflow*). In tal caso ritorna al punto 4.
6. Fine.

Non si propongono esercizi di normalizzazione perché sono inglobati negli esercizi della prossima sezione.

2.3.3.6 La conversione Decimale-FP

La conversione da numero decimale a numero in virgola mobile nasconde qualche problema in più rispetto alla conversione da virgola mobile a decimale, vista nella sezione 2.3.3.2, dato che ingloba l'arrotondamento e la normalizzazione.

Si è visto, infatti, nelle precedenti sezioni, che né l'arrotondamento né la normalizzazione sono azioni assolutamente prive di difficoltà.

Di fatto, la conversione decimale-virgola mobile prevede le seguenti azioni fondamentali:

1. la conversione del numero reale in binario;
2. la definizione del segno;
3. la definizione dell'esponento in eccesso 127;
4. la normalizzazione della mantissa;
5. la rappresentazione del numero in virgola mobile.

Di seguito si propongono alcuni esercizi di conversione che si ritengono essere utili a coprire i casi fondamentali. Si consiglia, pertanto, un loro attento studio.

Inoltre, al fine della conversione, si adotteranno le relazioni 2.17, 2.16 e 2.19. Anche la figura 2.4 risulterà utile.

Esercizio - ◊◊◊ Conversione con condizione di zero

Si vuole convertire il numero reale 0.0 nell'equivalente numero di macchina in virgola mobile in singola precisione.

Soluzione

Il bit di segno non ricopre, in questo contesto, un ruolo fondamentale. Viene comunque posto a 0, in modo che il numero, pur trattandosi dello 0, quindi né positivo né negativo, risulti positivo[12].

Il passo successivo, trattandosi di un'eccezione, consiste nell'applicazione della condizione di zero indicata dalla 2.19 a pagina 105, per cui si pone $f = 0$ ed $e = 0$.

Il numero in virgola mobile che ne emerge è il seguente:

$$00000000\ 00000000\ 00000000\ 00000000 \tag{2.28}$$

che rappresenta esattamente il numero reale 0.0 espresso in virgola mobile.

L'esercizio appena concluso è evidentemente un'eccezione e si è voluto presentarlo proprio all'inizio per sollevare l'argomento prima di affrontare i successivi esercizi con maggior metodo.

Già il prossimo esercizio verrà affrontato in tutt'altro modo e in maniera molto più canonica. La risoluzione dell'esercizio proposto utilizza molte delle tecniche studiate nelle precedenti sezioni, che lo studente dovrebbe essere in grado di utilizzare con sicurezza, e rappresenta una conversione di tipo assolutamente generale.

Esercizio - ◊◊♦ Conversione decimale-virgola mobile

Si vuole convertire il numero reale -123.456 nell'equivalente numero di macchina in virgola mobile in singola precisione.

Soluzione

Il bit di segno deve essere posto a 1, in modo tale da ottenere la seguente equivalenza: $(-1)^s = -1$.

Il modulo del numero è formato da una parte intera pari a 123 e da una parte decimale pari a 0.456.

Le due parti vanno convertite separatamente con i due metodi già discussi nella sezione 2.3.1: la parte intera va convertita mediante divisioni ripetute

---

[12] Il problema del segno posto davanti allo 0, non viene qui trattato dettagliatamente. Lo standard IEEE 754 affronta comunque il problema in maniera rigorosa dissipando qualsiasi ambiguità. Esistono ineluttabilmente due rappresentazioni dello 0, ossia $+0$ e -0. Lo standard impone l'uguaglianza $+0 = -0$, che risolve gran parte dei problemi, ma non tutti, soprattutto in fase di moltiplicazione e divisione coinvolgenti dei NaN (*Not a Number*), ove i segni vengono mantenuti. Lo studio di tali situazioni implica, però, la conoscenza di una parte dello standard che si decide di non analizzare nelle presenti pagine, per cui ci si accontenterà di trattare lo 0 come se fosse senza segno.

per due e la parte decimale va convertita mediante moltiplicazioni ripetute per due.

Dalla conversione della parte intera si ottiene:

123	1
61	1
30	0
15	1
7	1
3	1
1	1
0	

Il risultato della conversione è quindi: $123_{10} = 1111011_2$. Si noti che è stato convertito il modulo della parte intera, senza tener conto del segno, e che il numero non è espresso in complemento a due ma in notazione *unsigned*.

La conversione della parte decimale avviene mediante moltiplicazioni successive e produce:

$0.456 \cdot 2 = 0.912$	0
$0.912 \cdot 2 = 1.824$	1
$0.824 \cdot 2 = 1.648$	1
$0.648 \cdot 2 = 1.296$	1
$0.296 \cdot 2 = 0.592$	0
$0.592 \cdot 2 = 1.184$	1
$0.184 \cdot 2 = 0.368$	0
$0.368 \cdot 2 = 0.736$	0
$0.736 \cdot 2 = 1.472$	1
$0.472 \cdot 2 = 0.944$	0
$0.944 \cdot 2 = 1.888$	1
$0.888 \cdot 2 = 1.776$	1
$0.776 \cdot 2 = 1.552$	1
$0.552 \cdot 2 = 1.104$	1
$0.104 \cdot 2 = 0.208$	0
$0.208 \cdot 2 = 0.416$	0
$0.416 \cdot 2 = 0.832$	0
$0.832 \cdot 2 = 1.664$	1
$0.664 \cdot 2 = 1.328$	1

Non si ritiene utile proseguire, dato che si è arrivati già a 25 bit di mantissa. La parte frazionaria non assume quindi un valore esatto, perché non rappresentabile nei bit a disposizione (ovvero $23 + 1 - 7 = 17$), ma il valore $0111010010111100011_2 = 0.45599365234375_{10}$.

Il numero da convertire è quindi $1111011.0111010010111100011_2$.

Per effetto dell'arrotondamento i due bit meno significativi vanno persi ed il nuovo numero da normalizzare è $1111011.01110100101111000_2$.

La normalizzazione, a questo punto diventa banale:

$$1111011.01110100101111000 = 1.11101101110100101111000 \cdot 2^6 \qquad (2.29)$$

Siccome l'esponente va espresso in eccesso 127 esso assume il valore:

$$e - 127 = 6 \quad \rightarrow \quad e = 133 \tag{2.30}$$

ossia $e = 10000101_2$.

Togliendo il *hidden bit*, l'intero numero in virgola mobile diventa quindi:

$$-123.456_{10} \approx 1\ 10000101\ 11101101110100101111000 \tag{2.31}$$

evidenziando il segno, l'esponente e la mantissa. Il numero espresso in esadecimale diventa $0xC2F6E978$.

E' anche piuttosto banale calcolare l'errore assoluto commesso nella rappresentazione:

$$\varepsilon_a = fp(x) - x = 123.45599365234375 - 123.456 \approx -6.35 \cdot 10^{-6} \tag{2.32}$$

come pure l'errore relativo:

$$\varepsilon_r = \frac{\varepsilon_a}{x} = \frac{-6.35 \cdot 10^{-6}}{123.456} \approx 5.14 \cdot 10^{-8} \tag{2.33}$$

Come si vede l'errore è relativamente piccolo ed in molti casi può essere considerato ininfluente all'atto pratico. Nei casi in cui si debba avere una maggior precisione si può optare per una doppia precisione, diminuendo ulteriormente l'errore. L'importante è sottolineare che non è possibile rappresentare in maniera esatta un qualsiasi numero razionale, anche se non periodico, se lo si deve rappresentare con un numero di bit finito. Di ciò si deve tenere debitamente conto.

Il prossimo esercizio è un esempio di conversione esatta di un numero reale in base 10 nell'equivalente numero in virgola mobile.

Esercizio ◊◊◊ Una conversione esatta

Si vuole convertire il numero reale 12.46875 nell'equivalente numero di macchina in virgola mobile in singola precisione.

Soluzione

Il numero non è stato scelto a caso, lo si dice subito. La sua parte decimale è formata in modo tale che sia esatta somma di un polinomio formato da termini di potenza di due negative. La presente è effettivamente condizione necessaria e sufficiente affinché un numero decimale sia esattamente rappresentabile in virgola mobile.

Il numero da convertire è positivo, per cui si ha $s = 0$.

Inoltre, esso è formato dalla parte intera 12 e dalla parte decimale 0.46875.

Dalla conversione della parte intera si ottiene: ovvero $12_{10} = 1100_2$.

12	0
6	0
3	1
1	1
0	

Il presente caso è piuttosto semplice, dato che il numero da convertire è ≥ 1. In tal caso la cifra più significativa della parte intera del numero convertito è sicuramente un 1. Detta situazione è relativamente semplice da gestire perché tale 1 diventerà il *hidden bit* della mantissa, come si vedrà fra poco.

La conversione della parte decimale avviene, come al solito, mediante moltiplicazioni successive per la base di arrivo e produce il risultato riprodotto nei sottostanti calcoli:

$$\begin{array}{r|l} 0.46875 \cdot 2 = 0.9375 & 0 \\ 0.9375 \cdot 2 = 1.875 & 1 \\ 0.875 \cdot 2 = 1.75 & 1 \\ 0.75 \cdot 2 = 1.5 & 1 \\ 0.5 \cdot 2 = 1.0 & 1 \end{array}$$

ovvero $0.46875_{10} = 01111_2$.

Si noti che in questo caso non si ha alcun troncamento della mantissa che, anzi, viene agevolmente convertita in maniera esatta con soli 5 bit. Ciò significa che si dovranno aggiungere zeri non significativi fino al 23esimo bit.

Il numero così ottenuto e che va convertito in virgola mobile a 32 bit in base 2 eccesso 127 è quindi il seguente:

$$12.46875_{10} = 1100.01111_2$$

che è assolutamente equivalente a

$$12.46875_{10} = 1100.0111100000000000000_2 \tag{2.34}$$

che è la rappresentazione che permetterà di utilizzare tutti i 23 bit della mantissa[13].

Il numero indicato in 2.34 deve ora essere normalizzato, dopo aver tolto il *hidden bit*:

$$100.0111100000000000000_2 = 0.10001111000000000000000 \cdot 2^3 \tag{2.35}$$

Si noti che l'arrotondamento viene effettuato per difetto, dato che la cifra troncata più significativa non è 1.

L'esponente in eccesso 127 diventa

$$0.10001111 \cdot 2^{130} \tag{2.36}$$

e siccome $130_1 0 = 10000010_2$, il numero convertito diventa:

$$12.46875_{10} = 0\ 10000010\ 10001111000000000000000_2 \tag{2.37}$$

che rappresenta in maniera esatta il numero 12.46875_{10}, ove il bit di segno, gli 8 bit di esponente e i 23 bit di mantissa sono opportunamente evidenziati.

Se il numero da convertire è ≥ 1 la normalizzazione avviene sempre mediante shift a destra e rappresenta una situazione privilegiata. Se, invece, il

[13]Contando i bit della mantissa espressa dalla 2.34 si ottengono 24 bit e non 23. Si deve però togliere il *hidden bit*, ovvero il bit più significativo posto a 1.

numero da convertire è < 1, si hanno due casi distinti: 1) il numero è già normalizzato; 2) il numero va normalizzato mediante shift a sinistra.

Il prossimo esercizio chiarirà quanto detto.

Esercizio - ◊◊◊ Normalizzazione di numero razionale

Si vuole convertire il numero reale 0.46875 nell'equivalente numero di macchina in virgola mobile in singola precisione.

Soluzione

Tale numero è stato scelto al fine di poter risparmiare ulteriori conversioni: la parte intera vale 0 e la parte frazionaria è identica a quella dell'esercizio precedente, per cui $0.46875_{10} = 01111_2$.

Si deve quindi normalizzare il numero 0.01111_2 dato che esso è minore di $1/b$ ovvero 0.5. Ciò avviene shiftando a sinistra di 2 posizioni il numero, con conseguente diminuzione dell'esponente di 2 unità, in modo tale che il primo 1 venga posizionato alla sinistra della virgola. Tale bit, che ora è parte intera ed è pari a 1, diventa il *hidden bit* e viene tolto, per cui il numero normalizzato diventa

$$111_2 \cdot 2^{-2} \tag{2.38}$$

che in eccesso 127 diventa

$$111_2 \cdot^{125} \tag{2.39}$$

Essendo $125_1 0 = 01111101_2$ il numero in virgola mobile definitivamente convertito diventa:

$$0.46875_{10} = 0\ 01111101\ 11100000000000000000000_2 \tag{2.40}$$

Naturalmente non ha senso, per gli ultimi due esercizi, calcolare l'errore di approssimazione, dato che non si è operata alcuna approssimazione e si ha $\varepsilon_a = 0$ e $\varepsilon_r = 0$.

2.3.4 Il tipo char

I caratteri ai quali fanno riferimento le variabili di tipo `char` sono rappresentati nella tabella ASCII (vedi Appendice A) formata da 128 simboli. Esiste anche una tabella ASCII estesa formata da 256 simboli, che, però, non viene trattata nei presenti appunti.

Il tipo carattere è formato da 8 bit (1 byte), mediante i quali è possibile memorizzare 2^8 codici diversi, ovvero, in modalità *unsigned*, tutti i valori compresi fra 0 e $2^8 - 1 = 255$. In modalità *signed* i valori rappresentabili sono compresi fra -2^7 e $2^7 - 1$, ovvero fra -128 e +127. In ambedue i casi il tipo `char` ha cardinalità 256.

Il tipo `char` dovrà essere usato ogni qualvolta si vuole far riferimento ad uno dei simboli presentati nella tabella ASCII, sia che detto simbolo sia *printable*, ossia visibile, quindi avente valore ≥ 32 (space), oppure no. Se la variabile di tipo `char` è usata per accedere e gestire caratteri, non è necessaria alcuna variazione dello specificatore di tipo, ed un esempio di definizione di variabile potrebbe essere la seguente:

Listing 2.1: Definizione di variabile di tipo char

```
char carattere;
```

Esiste, però, anche la possibilità di utilizzare il tipo `char` in altri contesti, non necessariamente legati alla visualizzazione e gestione di caratteri. Si può utilizzare come intero con cardinalità limitata a 256, ovvero in tutte le operazioni aritmetiche con *range* fra -128 e +127 (modalità *signed*) oppure fra 0 e +255 (modalità *unsigned*).

Nel primo caso si può assimilare il tipo ad un sottoinsieme dei numeri relativi, nel secondo ad un sottoinsieme dei numeri naturali.

La seguente è una tabella riassuntiva dei diversi usi del tipo `char`:

Definizione	*Range*	
`char`	-128	+127
`signed char`	-128	+127
`unsigned char`	0	+255

Tabella 2.4: Possibili definizioni/dichiarazioni di char

E' possibile formulare ulteriori esempi di definizioni di variabile di tipo `char` corrette ed errate, come indicato nella sottostante tabella:

Corretto	Errato
`char` pippo;	`unsigned` pippo;
`signed char` pippo;	pippo `signed char`;
`unsigned char` pippo;	`signed` pippo;

Tabella 2.5: Ulteriori esempi di definizione di variabile ti tipo char

2.3.5 Il tipo int

Il tipo `int` serve a rappresentare variabili di tipo intero in un *range* più ampio rispetto a quello del tipo `char`. La sua effettiva estensione di campo dipende, però, dall'ambiente di sviluppo e dal sistema operativo.

Se il microprocessore da programmare è quello di un PC ospitante un sistema operativo a 32 o 64 bit, l'estensione tipica di una variabile di tipo `int` è di 32 bit, adatta a coprire un *range* compreso fra -2^{31} e $+2^{31}-1$, ovvero fra -2147483648 e +2147483647.

Se, invece, l'ambiente di sviluppo è quello dedicato ad un microcontrollore a 8 bit, sovente l'estensione è di soli 16 bit, adatta a coprire un *range* compreso fra -2^{15} e $+2^{15}-1$, ovvero fra -32768 e +32767. A meno di chiare indicazioni in senso contrario, nel prosieguo dei presenti appunti si supporrà implicito sempre il primo caso.

Lo studente potrebbe mostrarsi deluso per la, tutto sommato, scarsa estensione numerica del tipo `int`, anche nella sua versione a 32 bit. Esiste la possibilità di "dimensionare" il numero di bit del tipo `int` alle proprie esigenze,

aumentando, se necessario anche il numero di detti bit. Ciò avviene utilizzando gli *specificatori di tipo* `short`, `long` e `long long`, come nel sottostante specchietto:

Definizione	*Range*	
`short int`	-2^{15}	$+2^{15}-1$
`int`	-2^{31}	$+2^{31}-1$
`long int`	-2^{31}	$+2^{31}-1$
`long long int`	-2^{63}	$+2^{63}-1$

Tabella 2.6: Possibili specificatori del tipo int

Quindi uno `short int` (o semplicemente `short`) copre un *range* compreso fra -32768 e +32767; un `long int` (o semplicemente `long`) copre un *range* compreso fra -2147483648 e +2147483647; un `long long int` (o semplicemente `long long`) copre un *range* compreso fra -9223372036854775808 e +9223372036854775807 o, se si preferisce, circa $\pm 9 \cdot 10^{18}$.

Come il tipo `char`, anche il tipo `int` prevede l'uso degli specificatori *signed* ed *unsigned*. Una tabella riassuntiva delle estensioni in formato *unsigned* è la seguente:

Definizione	*Range*	
`unsigned short int`	0	$+2^{16}-1$
`unsigned int`	0	$+2^{32}-1$
`unsigned long int`	0	$+2^{32}-1$
`unsigned long long int`	0	$+2^{64}-1$

Tabella 2.7: Estensione del tipo unsigned int

2.3.6 I tipi float e double

I tipi `float` e `double` sono stati già trattati nella sezione 2.3.3.3 in maniera sufficientemente esaustiva. Si aggiunge unicamente che si tratta di formati utili a trattare numeri decimali di estensione relativamente bassa (`float`) o alta (`double`).

2.4 Un semplice esercizio

La scelta di un tipo di dato piuttosto che un altro dovrebbe essere sempre oggetto di riflessione. Il programmatore avrà cura, entro certi limiti, di fare una rapida analisi del tipo di dato che meglio si adatta al tipo di esigenza e scegliere di conseguenza. Scegliere tipi "abbondanti" non è sempre una buona politica:

appesantiscono i *data base* e rallentano i processi. Si veda, a tal proposito, il seguente

Esercizio - ◊◊◊ Scelta del tipo di dato

Si vuole creare un contatore che conti i secondi trascorsi dal *big bang* ad oggi. Si assuma che dall'esplosione primigenia alla mezzanotte del 31 dicembre 2013 siano passati 12.75 miliardi di anni. Quale fra i dati primitivi meglio si addice a tale uso e fino a quale data, approssimativamente, può continuare ad essere usato?

Soluzione

L'uso di contatori per memorizzare degli eventi temporali è molto frequente in informatica. Solitamente si prende una data di riferimento (le ore 0:00 del 1 gennaio 2000, ad esempio) e si contano i minuti o i secondi (a seconda della precisione voluta) che intercorrono fra il riferimento e l'evento. Tale tecnica permette di risparmiare spazio in memoria che, in certi casi, è utile ed efficiente.

Quindi la prima cosa da fare è valutare l'ammontare del numero di secondi intercorsi fra il *big bang* e la mezzanotte del 31 dicembre 2013. Con precisione adeguata alla presente facezia si può scrivere:

$$bbsec = 12.75 \cdot 10^9 \cdot 365.25 \cdot 24 \cdot 60 \cdot 60 \; sec \qquad (2.41)$$

che equivale a

$$bbsec = 402359400 \cdot 10^9 = 4.023594 \cdot 10^{17} \; sec \qquad (2.42)$$

Ora si può valutare quale tipo sia più adatto ad essere utilizzato come contatore.

E' necessario usare un tipo che permetta la rappresentazione di numeri a 18 cifre. Si possono subito scartare i tipi `char`, `int` e `float`, che sono assolutamente insufficienti. Il tipo `double` può rappresentare numeri fino a circa 16 cifre ($\sim 4.5 \cdot 10^{15}$) ed è quindi anch'esso insufficiente.

Il tipo `long long` appare, invece, adatto alla bisogna, potendo rappresentare, addirittura in formato con segno, numeri positivi a 19 cifre ($\pm 9 \cdot 10^{18}$). Anzi, verrebbe sfruttato meno di un ventesimo della capacità del `long long` e meno di un quarantesimo di un `unsigned long long`.

A questo punto pare assolutamente insensato calcolare la data approssimativa di fine utilizzo di detta variabile.

2.5 La costante

La costante assume nei linguaggi ad alto livello lo stesso significato che assume in matematica: un valore costante appartenente ad un determinato insieme. A detto valore viene abbinato un identificatore in modo tale che possa essere più facilmente identificabile il significato che un determinato valore costante assume in un contesto. Si supponga, ad esempio, di utilizzare il valore 7 in un determinato programma e che detto valore rappresenti il numero dei giorni di

una settimana, il numero di termini di una serie aritmetica e ancora il numero di cifre significative da attribuire al valore approssimato di π.

Se all'interno di un determinato programma il valore 7 rappresenta tutto quanto appena esposto, c'è il rischio di far confusione nel momento in cui si desidera aumentare il numero di cifre significative del π portandolo a 8: si rischia di aumentare anche il numero di termini della serie oppure, peggio, ancora, di stabilire che i giorni della settimana sono 8.

Se invece, si usano tre identificatori diversi, aventi tutti il valore 7, tale rischio diminuisce sensibilmente. Si supponga che i tre identificatori si chiamino `kGiorniSettimana`, `kNumeroTermini` e `kCifreSignificative`. Nel momento in cui si desidera modificare il numero di cifre significative del π, sarà sufficiente modificare il valore attribuito a `kCifreSignificative`.

2.5.1 La dichiarazione di costante

In linguaggio C la costante va dichiarata in maniera del tutto simile alla variabile. Un esempio di dichiarazione di costante è il seguente:

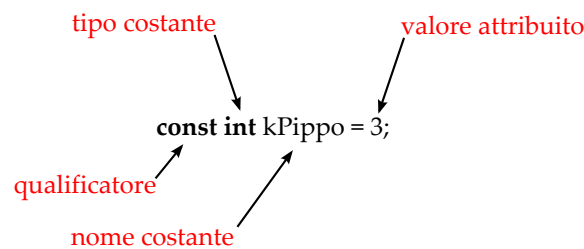



Figura 2.7: Esempio di dichiarazione di costante

Le due principali differenze con la definizione di variabile sono date dal *qualificatore di costante* posto davanti al tipo e dal fatto che l'attribuzione del valore alla costante in fase di dichiarazione *non è opzionale* ma obbligatorio (vedi la sezione 3.5).

Inoltre, una buona usanza è data dall'anteporre al nome mnemonico della costante la lettera minuscola "**k**", ad indicare che si tratta, appunto, di una costante.

Si noti che si è sempre utilizzato il verbo "dichiarare" e non "definire". La distinzione è piuttosto importante soprattutto per gli elettronici/telecomunicazionisti piuttosto che per gli informatici.

Quando la costante viene dichiarata all'interno di una funzione non viene riservato alcuno spazio nello *stack*. Essa viene memorizzata nella stessa area di memoria ove risiede il codice, quindi nello *heap*[14] nel caso in cui il programma sia lanciato dal sistema operativo di un PC oppure nella memoria di programma se il sistema è di tipo *embedded*. In ambedue i casi la *dichiarazione di costante occupa comunque dello spazio in memoria*.

La cosa può diventare fastidiosa nel momento in cui la costante non sia una sola ma un *array* di costanti (si pensi, ad esempio, ad una tabella più o meno

[14] La differenza fra *stack* e *heap* verrà trattata nei prossimi capitoli.

grande) e debba trovare spazio nella memoria di programma di un microcontrollore con memoria di piccole dimensioni (4-8KB). In tal caso può tornare utile *definire* la costante mediante la direttiva `#define`.

2.5.2 La direttiva #*define*

La definizione di costante non deve essere confusa con la definizione di variabile. La seconda *alloca* spazio nello *stack*, mentre la prima utilizza la direttiva `#define`. Questa direttiva del preprocessore è tanto semplice quanto potente.

Definizione 4 (Direttiva #*define*)**.**
La direttiva `#define` *abbina una qualsiasi successione di caratteri ad un identificatore. Prima della compilazione, il preprocessore sostituisce, nel programma, l'identificatore con la successione di caratteri abbinata.*

Un esempio d'uso della direttiva è il seguente:

Listing 2.2: Esempio di #*define*

```
#define kLuglio 7
```

L'identificatore è quello che segue la direttiva, mentre la successione di caratteri segue l'identificatore e termina con la fine riga, ovvero iniziando a scrivere su una nuova riga. Nell'esempio citato, prima della compilazione il preprocessore sostituisce il valore 7 ogni qualvolta incontra l'identificatore `kLuglio`. Questa semplice sostituzione di caratteri è eccezionalmente potente, ma *va presa alla lettera*. Nel prossimo capitolo si forniranno alcuni esempi che chiariranno meglio tale concetto.

2.6 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 2.

Capitolo 3

L'assegnazione

L'assegnazione è l'istruzione fondamentale di ciascun linguaggio di programmazione e, in quanto tale, merita una definizione tutta sua:

Definizione 5 (Assegnazione)**.**
L'assegnazione è l'operazione mediante la quale si attribuisce un valore ad una variabile.

Un esempio di assegnazione (definizione di variabile inclusa) potrebbe essere la seguente:

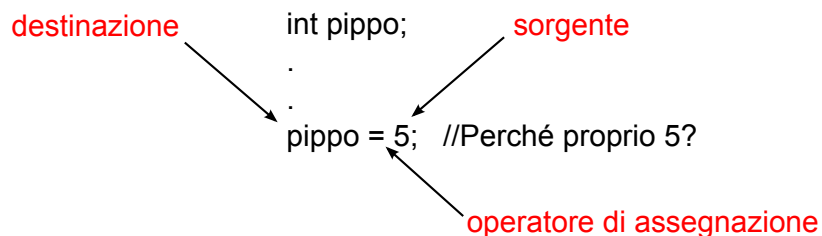



Figura 3.1: Istruzione di assegnazione

Anche in questo caso una rapida analisi non guasta. L'espressione di assegnazione in questo caso coincide con l'istruzione omonima ed è composta da tre elementi[1]:

- da una **destinazione**;
- da una **sorgente**;
- da un **operatore di assegnazione**.

[1]Si consiglia allo studente di prestare, in generale, grande attenzione alla terminologia usata: l'insegnante ha la tendenza ad essere piuttosto preciso nell'uso dei termini, per cui se ad una determinata parola vengono assegnati, da parte dello studente e dell'insegnante, significati diversi la comunicazione risulta essere inefficace (trad.: lo studente non capisce).

L'espressione/istruzione è chiusa dal terminatore (;).

La **destinazione** deve essere sempre e solamente una variabile già dichiarata o definita precedentemente (o, come si vedrà, contestualmente). Non può essere altro.

Più articolata è la definizione della **sorgente**. La sorgente può essere:

- una costante;
- una variabile;
- una funzione[2];
- un'espressione formata da uno o più dei precedenti elementi.

All'**operatore di assegnazione** conviene, infine, dedicare un'apposita sezione.

3.1 L'operatore di assegnazione

L'operatore di assegnazione è fonte di più di qualche incomprensione da parte dello studente. La prima e più ovvia osservazione da fare è relativa all'utilizzo del simbolo "=". In matematica il simbolo "=" è un operatore di uguaglianza ed indica la formale e sostanziale uguaglianza fra il membro di sinistra ed il membro di destra di un'espressione matematica. Nel linguaggio C, però, detto simbolo è utilizzato per rappresentare l'operatore di assegnazione, non quello di ugualianza. Lo studente lo deve tener ben presente.

Un altro spunto di riflessione viene fornito da certe assegnazioni in cui la sorgente è un'espressione, in particolare quelle che hanno la seguente forma:

Listing 3.1: Assegnazione particolare

```
pippo = pippo + 7;      //Assegnazione che suscita perplessita'
```

Un matematico privo di nozioni di Informatica resterebbe perplesso di fronte ad una simile espressione. La perplessità nasce dal fatto che l'operatore di assegnazione viene interpretato come operatore di uguaglianza. In quest'ottica l'espressione 3.1 è assolutamente assurda ed inaccettabile, dato che il membro di destra della 3.1 è palesemente diverso dal membro di sinistra.

Se, invece, interpretassimo la 3.1 nel seguente modo

$$\text{pippo} \leftarrow \text{pippo} + 7; \quad \text{//Assegnazione che suscita perplessità}$$

potrebbe diventare più comprensibile. E potrebbe essere ancora più comprensibile se scritta nella presente forma:

$$\text{pippo}_{t_2} \leftarrow \text{pippo}_{t_1} + 7; \quad \text{//Assegnazione che suscita perplessità} \tag{3.1}$$

Al tempo t_1 la variabile `pippo` ha un certo valore. Tale valore viene aumentato di 7 ed il risultato di tale somma viene assegnato, nel tempo t_2, alla variabile `pippo` come nuovo valore che essa deve assumere.

La confusione nasce dal fatto che le equazioni algebriche non sono funzioni del tempo, mentre le assegnazioni sono delle azioni che vengono eseguite in funzione del tempo, per giunta discreto. L'argomento è già stato trattato

[2] Alcune funzioni particolari verranno trattate nei prossimi capitoli. Per il momento lo studente può pensare alle funzioni matematiche, con le quali ci sono molte similitudini.

nella documentazione relativa agli algoritmi, redatta dallo stesso autore della presente, e chi ne fosse in possesso e nutrisse ancora dei dubbi, farebbe bene a (ri)leggersi tale trattazione.

Un ultimo spunto di riflessione è sintetizzato dalla seguente assegnazione:

Listing 3.2: Che fine fa pluto?

```
pippo = pluto;  //Che fine fa pluto?
```

Frequentemente lo studente alle prime esperienze con i linguaggi di programmazione avanza un dubbio: "Ho capito che `pippo` assume il valore di `pluto`, ma `pluto` che fine fa?". La risposta è semplicissima: `pluto` rimane assolutamente inalterato rispetto al valore che possedeva prima della assegnazione.

3.2 Il *type matching*

La definizione 5 a pagina 123 definisce l'assegnazione, ma non le regole che la sorreggono. La regola principale che qualsiasi assegnazione deve osservare è quella del *type matching*: il tipo sorgente deve essere ammissibile dal tipo destinazione. La frase è banale, ma nasconde qualche insidia.

La situazione più semplice è quella che prevede lo stesso tipo sia per la sorgente che la destinazione. Si valuti il seguente codice[3]:

Listing 3.3: Che valore assume pippo?

```
char pippo;

//Che valore assume pippo nelle
//sottostanti quattro istruzioni?
pippo = 12;
pippo = 12+56;
pippo = 54/12;
pippo = 54+74;
```

La prima assegnazione non presenta nessun tipo di difficoltà: la destinazione è di tipo `char` e la sorgente pure (perché?), per cui c'è assoluto *type matching*. Il tipo alla sinistra dell'operatore di assegnazione ed il tipo alla destra dell'operatore sono identici. A `pippo` viene quindi assegnato il valore intero 12.

Anche la seconda assegnazione non crea problemi: la sorgente è rappresentata da un'espressione aritmetica, più precisamente da una somma fra due costanti. A `pippo` viene assegnato il valore risultante da detta somma, ossia 68.

Qualche primo problema nasce dalla terza istruzione. L'operazione aritmetica 54/12 produce un quoziente 4 con resto 6, per cui in `pippo` verrà caricato 4 e non 4.5, come qualche precipitoso studente potrebbe immaginare. Essendo la sorgente formata da un'espressione di costanti intere, il risultato sarà una costante intera e non decimale.

La quarta espressione è piuttosto infida. In `pippo` viene caricato il valore -128. Perché? Lo studente "cartesiano" dovrebbe aver notato che...

[3]Si danno per scontate, momentaneamente, le quattro operazioni artitmetiche.

... che 54+74 = 128 e 128 non è rappresentabile mediante una variabile di tipo `char`. Il valore 128 in binario si scrive nel seguente modo

1 0 0 0 0 0 0 0

ed essendo tale valore assunto in complemento a due viene interpretato come valore negativo, dato che il bit più significativo vale 1.

Siccome in complemento a due il numero 10000000_2 vale proprio -128_{10}, questo è il valore che viene memorizzato nella variabile `pippo`. Si noti che se `pippo` fosse stato definito come `unsigned char`, il risultato sarebbe stato memorizzato correttamente, dato che `pippo` avrebbe potuto contenere senza ambiguità il valore +128.

Ragionamenti del tutto analoghi si possono fare per i tipi `int`, `float` e `double`.

3.3 Assegnazioni fra tipi differenti

Le cose si complicano leggermente quando c'è diversità di tipo fra sorgente e destinazione. Siccome i quattro tipi primitivi sono tutti numerici, è possibile stabilire una correlazione gerarchica fra essi. Il tipo `char` è "contenuto" nel tipo `int`, nel senso che un numero intero che sia rappresentabile mediante un `char` lo è tanto più mediante un `int`. E siccome un numero intero può essere rappresentato anche in notazione decimale, analogamente un valore di tipo `int` può essere caricato senza problemi in una variabile di tipo `float` o `duoble`.

La correlazione gerarchica fra tipi numerici è evidenziata mediante diagrammi di Eulero-Venn in fig. 3.2.

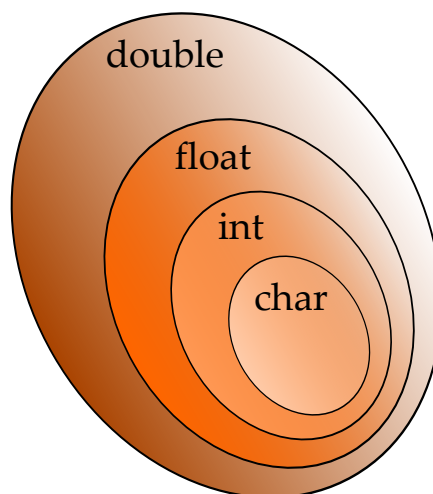



Figura 3.2: Correlazione fra i quattro tipi primitivi

Per contro, non si può pretendere di memorizzare un numero di tipo `int`, ad esempio di 4 cifre, in una variabile di tipo `char`, né si può pretendere di memorizzare un numero decimale, ad esempio 3.141592, in una variabile di tipo `int`, che non permette la rappresentazione in virgola mobile.

Quando si eseguono assegnazioni permesse fra tipi differenti, si possono avere due casi distinti: assegnazioni senza perdita di informazione e con perdita di informazione.

3.3.1 Assegnazioni senza perdita d'informazione

Il primo dei due tipi di assegnazione appena citati è naturalmente da preferirsi. In questo caso l'assegnazione avviene senza alcuna perdita di informazione. Detto in altri termini, un po' semplicistici, non si ha "perdita" di cifre durante l'assegnazione. Si vedano gli esempi sottostanti:

Listing 3.4: Assegnazioni permesse

```
char dotto;
int brontolo;
float pisolo;
double eolo;

dotto = 56; brontolo = 1234; pisolo = 0.5; eolo = 0.8;
brontolo = dotto;              //brontolo = 56;
eolo = pisolo;                 //eolo = 0.5
pisolo = dotto + brontolo;     //pisolo = 112.0
pisolo = eolo + pisolo;        //Warning
```

Tutte le istruzioni, meno l'ultima, rispettano il rapporto gerarchico illustrato in fig. 3.2, per cui non si ha perdita di informazione. L'ultima istruzione produce un *Warning* da parte del compilatore, ma nessuna perdita di informazione, dato che il risultato (112.0 + 0.5) è rappresentabile mediante un `float`.

3.3.2 Assegnazioni con perdita d'informazione

Di seguito vengono proposte alcune assegnazioni in cui, invece, si ha perdita di informazione. Ciò significa che nel passaggio dalla sorgente alla destinazione si perdono alcuni bit, in seguito ad una (ulteriore) approssimazione. Si vedano i seguenti esempi:

Listing 3.5: Assegnazioni con perdita d'informazione

```
char dotto;
int brontolo;
float pisolo;
double eolo;

dotto = 56; brontolo = 1234; pisolo = 0.5; eolo = 0.8;
pisolo = brontolo/dotto;       //Perdita d'informazione
pisolo = eolo;                 //Perdita d'informazione
```

Le ultime due istruzioni vanno evitate perché implicano una perdita di informazione. Il compilatore lo notifica attraverso la segnalazione di due *Warning*.

Non è però facilissimo capire dove e perché si ha perdita di informazione. Apparentemente la variabile `pisolo`, definita di tipo `float` è assolutamente in grado di contenere e rappresentare il risultato del rapporto 1234/56, come

pure la stessa variabile `pisolo` è in grado, apparentemente, di contenere un valore decimale normalissimo come 0.8.

Entrambe le affermazioni sono false.

Esaminiamo la prima delle due assegnazioni: vìola il *type matching*. La sorgente è un rapporto fra interi e come tale verrà eseguita. *Dopo* che la divisione sarà stata eseguita (e il quoziente sarà necessariamente intero), verrà assegnato a `pisolo` il risultato dell'operazione fra interi.

Più in particolare avviene quanto segue:

$$\frac{\text{brontolo}}{\text{dotto}} = \frac{1234}{56} = 22 \quad (3.2)$$

$$\text{pisolo} \leftarrow 22 \quad (3.3)$$

La divisione prodotta in 3.2 non produce il risultato decimale 22.0357, ma il risultato intero 22. Si può evitare l'inconveniente, senza dover cambiare il tipo delle variabili, ad esempio, nel seguente modo:

Listing 3.6: Divisione senza perdita d'informazione

```
pisolo = (brontolo+0.0)/dotto;  //Divisione OK
```

Si nota che a `brontolo` è stato sommato il valore decimale 0.0, trasformando, in tal modo, il numeratore in valore decimale. In questo caso il compilatore esegue[4] la somma uniformandosi al tipo della destinazione, non essendo definito il tipo della costante 0.0.

Il motivo per cui si ha perdita di informazione nella seconda espressione è ancora più subdolo. Anche perché `pisolo` accetta senza problemi il valore 0.5, per cui dovrebbe accettare analogamente un valore del tutto simile come 0.8.

In realta, come già visto nella sezione 2.3.3.1, il numero 0.8 non è per nulla simile al numero 0.5. Quest'ultimo è infatti esprimibile come potenza di 2 (ovvero 2^{-1}), mentre, in base due, il numero 0.8 è un numero periodico. Ciò significa che la variabile `eolo` contiene un valore approssimato di 0.8 e quando tale valore approssimato viene trasferito da una variabile in virgola mobile in doppia precisione (`eolo`) ad una in singola precisione (`pisolo`) si ha necessariamente perdita di informazione dovuta alla seconda approssimazione.

Come si vede, quindi, anche la più banale delle istruzioni, l'assegnazione, necessita sempre di un atteggiamento vigile e critico per essere usata con perizia. Al fine di sottolineare tale atteggiamento, si propone il seguente

Esercizio - ◊◊◊ Assegnazione con perdita d'informazione

Si calcoli l'errore assoluto e l'errore percentuale commesso nella assegnazione `pisolo = eolo;` del codice 3.5 nella pagina precedente.

Soluzione

Il numero 0.8, contenuto nella variabile `eolo` (di tipo `double`) è molto vicino a 1. Ciò significa banalmente che il valore dell'esponente non ha necessità di essere approssimato.

[4] E' evidente che la "esecuzione" vera e propria avviene ad opera del micrprocessore e non del compilatore. Ma il microprocessore esegue semplicemente quello che gli "prepara" il compilatore. E' quindi il compilatore che si uniforma, non il microprocessore.

L'approssimazione ha luogo nel passaggio dalla mantissa a 52 bit del `double` alla mantissa a 23 bit del `float`. Più precisamente si perde una quantità pari a

$$2^{-23} + 2^{-24} + 2^{-25} + ... + 2^{-51} \tag{3.4}$$

Il tipo `double` approssima il valore 0.8 a 0.80000000000000004, mentre il tipo `float` approssima lo stesso valore a 0.80000001.

L'errore assoluto vale quindi

$$0.80000000000000004 - 0.80000001 = -9.99999996 \cdot 10^{-9} \tag{3.5}$$

mentre l'errore percentuale vale

$$\frac{-9.99999996 \cdot 10^{-9}}{0.80000000000000004} \cdot 100 \approx -0.00000125\% \tag{3.6}$$

3.4 L'inizializzazione delle variabili

"Le variabili devono essere inizializzate prima di essere utilizzate". Questa frase viene scandita da tutti gli insegnanti di Informatica quando parlano di variabili. "Inizializzare" significa "attribuire un valore noto[5]".

Quindi, prima di essere "utilizzate", le variabili devono avere un valore calcolabile. Quando la variabile viene definita, invece, essa non ha un valore noto[6], ma casuale, sconosciuto e non calcolabile. Si tratta di capire con precisione cosa si intende con il termine "utilizzate". A tal fine può essere utile fare un paragone con l'ambiente matematico.

Si esamini la seguente equazione/assegnazione:

$$y = x + 3 \tag{3.7}$$

Normalmente si intende che la variabile x sia la variabile indipendente e y la variabile dipendente. Ciò significa che affinché si possa calcolare il valore della variabile y è necessario che sia noto il valore della variabile x.

Alla luce dei ragionamenti appena fatti, la frase di apertura della presente sezione assume un significato più circostanziato. Sia x che y sono variabili, ma affinché possa essere calcolato il valore di y è sufficiente che la sola variabile x possegga un valore noto.

E' anche piuttosto intuitivo che se il valore di x non dovesse essere definito ma "casuale", non sarebbe possibile calcolare il valore di y. Ed è anche intuitivo che se x è noto, non è necessario che y abbia anch'esso un valore noto prima del calcolo della 3.7, cioè non è necessario che sia inizializzata.

[5]"Valore noto" non significa che il programmatore lo conosce, ma che, volendo, potrebbe calcolarlo.

[6]In certi casi ciò non è vero. Un ambiente di sviluppo estremamente noto, il Visual C++ della Microsoft, inizializza automaticamente le variabili definite, attribuendo ad esse il valore, nel caso di varibili di tipo int, 0xCCCCCCCC. Questo particolarissimo valore permette all'ambiente di sviluppo di identificare con ottima approssimazione le variabili non inizializzate dal programmatore.

Quindi, per chiudere l'argomento, si specifica che tutte le variabili che *concorrono nel calcolo del valore della sorgente* devono essere inizializzate, ovvero possedere un valore noto, mentre quelle variabili che compiaiono solamente nella destinazione possono anche non esserlo.

3.5 Definizione e inizializzazione

Nella sezione 3.4 è stato affrontato l'argomento relativo all'inizializzazione delle variabili. E' stato chiarito che alcune fra le variabili dichiarate e definite devono essere inizializzate prima di essere usate. Il valore al quale dette variabili debbano essere inizializzate deve essere deciso dal programmatore, anche se, normalmente, lo zero è uno dei valori più gettonati.

Piuttosto ci si chiede qual è il momento migliore per inizializzarle.

Una convenzione alla quale spesso i programmatori si uniformano è quella dell'inizializzazione in fase di definizione. Essa viene eseguita come nell'esempio sottostante:

Listing 3.7: Inizializzazione durante la definizione

```
int y;          //Variabile dipendente
int x = 0;      //Variabile indipendente inizializzata

y = x+3;        //Uso della variabile inizializzata
```

La seconda riga evidenzia quale debba essere la sintassi della inizializzazione della variabile nella fase di definizione. Il vantaggio di una simile inizializzazione è dato dall'ordine: non è necessario spulciare riga per riga il programma alla ricerca dell'istruzione di assegnazione che inizializza la variabile, ma è sufficiente controllare la sezione di definizione/dichiarazione, che normalmente è posta a inizio funzione.

3.6 Il commento

La presente sezione tratta un argomento che molti studenti possono ritenere stucchevole ed inutile. Niente di più sbagliato!

Si ripropone un'espressione già vista qualche pagina addietro:

Listing 3.8: Utilità dei commenti

```
pippo = 5;      //Perche' proprio 5?
```

Spostiamo la nostra attenzione dall'assegnazione al commento: "Perché proprio 5?". Invece di rispondere direttamente si propone una variante al codice 3.8, come di seguito indicato:

Listing 3.9: Commento inutile

```
pippo = 5;      //Assegna a pippo il valore 5
```

Domanda: "Esiste nell'universo conosciuto qualcosa di più inutile del commento all'assegnazione 3.9?" Chiunque, anche il più neofita dei programmatori, è in grado di capire da sé che la 3.9 provvede ad assegnare il valore 5 a

`pippo`, non vi è alcuna necessità di sottolinearlo nel commento. Il valore di un commento simile è paragonabile ad un *Lorem Ipsum*[7].

Non si tratta di commentare *cosa* fa l'espressione 3.9, ma *perché* lo fa.

Ben più complessi e degni di commento potrebbero essere i motivi che inducono il programmatore ad assegnare il valore 5 a `pippo`. Naturalmente non vi è un motivo specifico nell'esempio citato (non ha senso ipotizzare un motivo, dato che la scelta dell'identificatore è dettata proprio dalla volontà di decontestualizzare il nome della variabile rispetto all'esempio), ma in un programma vero, bisogna sempre ricordarsi di commentare il *perché*, non il *cosa*.

Qualche ulteriore nozione sintattica.

Si è visto che il commento inizia con il metasimbolo "`//`". In tal caso tutti caratteri posti alla destra di detto simbolo verranno interpretati come commento, fino alla fine della linea. Un *carriage return* interromperà immediatamente il commento.

Un secondo modo per delimitare i commenti è dato dalla coppia di simboli `/*` ... `*/`. Tale soluzione permette di scrivere commenti su più righe come nel seguente esempio:

Listing 3.10: Altro modo di commentare

```
pippo = 5;      /* Questo e' un commento lungo lungo
                   che si estende su sole tre righe ma
                   potrebbe essere anche piu' esteso. */
```

Questo modo di commentare è spesso usato dai programmatori per "porre fra commento" gruppi di istruzioni che non devono essere momentaneamente eseguite, ad esempio durante il debugging di una certa parte di codice. E' utilissimo se usato per commentare le *direttive di definizione*, che verranno trattate nel capitolo dedicato alle direttive.

E per quanti non fossero convinti dell'utilità dei commenti è bene ricordare le ironiche parole di Ed Post, illustre programmatore, nel 1982, della *Graphic Software Systems*, il cui pensiero è ripreso in parte nell'Introduzione:

Real Programmers don't need comments: the code is obvious.

---

[7] *Per coloro i quali non lo sapessero, il **Lorem Ipsum** è un testo privo di significato, composto estrapolando a caso parole, talvolta storpiate, da un'opera di Cicerone, il "De finibus bonorum et malorum". E' utilizzato, sin dal 1500, dai tipografi (ma anche dai programmatori e dai Web designer) come riempitivo (spaceholder) per le prove di stampa. Di seguito è fornito il testo originale. Il grassetto evidenzia un classico **Lorem Ipsum**. "[] unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam eaque ipsa, quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt, explicabo. Nemo enim ipsam voluptatem, quia voluptas sit, aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos, qui ratione voluptatem sequi nesciunt, neque porro quisquam est, qui do**lorem ipsum**, quia **dolor sit, amet, consectetur, adipisci** v'**elit, sed** quia non numquam **eius modi tempor**a **incidunt, ut labore et dolore magn**am **aliqua**m quaerat voluptatem. **Ut enim ad minima veniam, quis nostru**m **exercitation**em **ullam co**rporis suscipit **labori**osam, **nisi ut aliquid ex ea commodi consequat**ur? **Quis aute**m vel eum **iure reprehenderit**, qui **in** ea **voluptate velit esse**, quam nihil molestiae consequatur, vel **illum**, qui **dolore**m **eu**m **fugiat**, quo voluptas **nulla pariatur**? At vero eos et accusamus et iusto odio dignissimos ducimus, qui blanditiis praesentium voluptatum deleniti atque corrupti, quos dolores et quas molestias **excepturi sint, obcaecati cupiditate non provident**, similique **sunt in culpa, qui officia deserunt mollit**ia **animi**, **id est laborum** et dolorum fuga.[]"* (cfr. [18])

3.7 Un esempio d'uso delle costanti

Nel capitolo precedente si è parlato di costanti, senza però avere ancora gli strumenti utili a spiegarne la funzionalità. Ora che il lettore si è impadronito dei concetti di assegnazione e di commento è possibile fare alcuni esempi.

Si ricordi l'esempio fatto nella sezione 2.5 e si valuti il seguente codice:

Listing 3.11: Cattivo uso delle costanti

```
ng = s*7;
...
nt = 7;
```

La prima riga calcola il numero di giorni (`ng`) trascorsi da una determinata data essendo noto il numero delle settimane trascorse (`s`). Dopo alcune istruzioni, non documentate, si attribuisce alla variabile `nt` il numero di termini di una serie aritmetica che dovrà essere calcolata.

Si supponga di essersi accorti, durante il debugging, che 7 termini sono troppo pochi per calcolare la serie con un minimo di precisione e di volerli aumentare a 10. C'è il rischio di confondersi e cambiare anche la costante 7 della prima riga, soprattutto se detta costante appare frequentemente nel codice. Inoltre, si dovrà controllare un alto numero di righe per verificare se tutte le costanti sono state modificate.

Si confronti ora il codice 3.11 con il seguente:

Listing 3.12: Corretto uso delle costanti

```
const int kGiorniSettimana = 7;
const int kNumeroTermini = 7;
...
ng = s*kGiorniSettimana;
...
nt = kNumeroTermini;
```

Se si intende modificare il numero di termini della serie, si modificherà solamente la dichiarazione della costante `kNumeroTermini`. Oltre ad essere molto più chiaro il codice, un simile approccio eviterà lunghe ricerche e annullerà la possibilità di errori durante la modifica.

3.8 Un esempio d'uso della direttiva *#define*

Gli stessi risultato ottenuti nella sezione precedente si potevano ottenere definendo le costanti nel seguente modo:

Listing 3.13: Possibile uso delle *#define*

```
#define kGiorniSettimana 7
#define kNumeroTermini 7
...
ng = s*kGiorniSettimana;
...
nt = kNumeroTermini;
```

Il vantaggio ottenuto è una minor occupazione del codice nella memoria di programma (o nello *heap*). In molti casi il vantaggio è assolutamente ininfluente, per cui usare la dichiarazione oppure la definizione di costante risulta essere praticamente la stessa cosa.

Vi sono, però, alcuni aspetti nell'uso della direttiva `#define` che vanno sottolineati, perché potenzialmente dannosi. Si veda il seguente codice:

Listing 3.14: Possibile uso delle *#define*

```
#define kGiorniSettimana 5+2
const int kgiornisettimana = 5+2;
...
ng = s*kgiornisettimana;
...
ng = s*kGiorniSettimana;
```

Le costanti `kGiorniSettimana` e `kgiornisettimana` sono dichiarati in tale modo per distinguere i giorni lavorativi da quelli festivi. Dal punto di vista pratico non vi è alcuna differenza: si tratta di stili di documentazione, peraltro molto diffusi. Alla stessa maniera, per indicare lo spazio occupato dalla stringa "pippo" si potrebbe scrivere `const int kLenPippo = 5+1;` per evidenziare che si tiene conto anche del terminatore.[8]

Le due righe di calcolo del numero di giorni nel codice 3.14 danno però due risultati molti diversi. Il primo calcolo risulta essere assolutamente corretto, mentre nel secondo caso il preprocessore, prima della compilazione, semplicemente *sostituisce* all'identificatore la successione di caratteri che lo segue, fornendo il seguente risultato:

Listing 3.15: Tipico errore nell'uso delle *#define*

```
ng = s*5+2;
```

Ovviamente, mancando la parentesi, il risultato è diverso da quello atteso.

Un altro errore frequente è rappresentato dal codice

Listing 3.16: Errore nell'uso del commento

```
#define kGiorniSettimana (5+2) //Le parentesi ci sono
...
ng = s*kGiorniSettimana;
```

Anche in questo caso c'è un evidente errore, che però per fortuna viene rilevato dal compilatore, trattandosi di un errore sintattico. Dopo la sostituzione si ottiene

Listing 3.17: Altro errore nell'uso delle *#define*

```
ng = s*(5+2) //Le parentesi ci sono;
```

ponendo di fatto il simbolo (;) dentro il commento, lasciando di conseguenza non terminata l'espressione.

Non si vuole scoraggiare lo studente dal commentare il codice: deve solo ricordarsi, quando commenta una `#define`, di farlo usando i delimitatori di commento `/* .. */` anzichè `//`.

Il codice, così corretto, diventa

[8]Per ulteriori dettagli sull'argomento si veda la sezione 9.4.

Listing 3.18: Uso corretto del commento nelle #*define*

```
#define kGiorniSettimana (5+2) /*Le parentesi ci sono*/
...
ng = s*kGiorniSettimana;
```

In tal modo il punto e virgola verrebbe salvato: in fase di compilazione il commento verrebbe semplicemente tolto ed il codice, punto e virgola compreso, compilato correttamente.

Vi sono ulteriori aspetti che andrebbero analizzati nell'uso della direttiva `#define`, come ad esempio la possibilità di definire istruzioni o gruppi di istruzioni anziché costanti, ma si tratta di un uso piuttosto bizzarro e più proteso alla "protezione" del codice piuttosto che all'effettivo utilizzo di una potenzialità del linguaggio C.

3.9 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 3.

Capitolo 4

I primi programmi

Abbiamo visto che la programmazione è un'arte,
perché richiede conoscenza, applicazione, abilità e ingegno,
ma soprattutto per la bellezza degli oggetti che produce.
Donald E. Knuth (1973)

Ora lo studente è in possesso del minimo indispensabile per scrivere il suo primo programma. Il "contenitore" di detto programma potrebbe essere simile al seguente:

Listing 4.1: Programma vuoto

```
//Funzione main
void    main(void)
{
   //Il corpo del main e' vuoto
}
```

Un simile programma vuoto è formato da una sola funzione chiamata `main`, che è l'*entry point* di qualsiasi programma scritto in linguaggio C. Esso fornisce già i primi spunti di riflessione.

4.1 Il *main*

Sostanzialmente sono tre i punti sui quali focalizzare l'attenzione:

- la parola chiave `main`;
- la parola chiave `void`;
- le parentesi graffe.

La parola chiave `main` significa "principale" e ciò spiega già molto. Il `main` è la funzione principale di un programma scritto in linguaggio C. Tutti i programmi scritti in C devono avere, evidente o nascosta[1], una funzione chiamata `main`. E siccome il linguaggio C è *case sensitive*, è importante che la parola chiave sia scritta in tutte lettere minuscole.

Fra parentesi tonde sono elencati gli argomenti della funzione. Siccome fra parentesi è posta la parola chiave `void` (vuoto), significa che il `main` non ha argomenti.

Come ogni funzione anche il `main` dovrebbe ritornare un valore, ma il programma di codice 4.1 non ritorna alcunché (`void`). Questo tipo di "contenitore" è usatissimo nei microcontrollori che gestiscono sistemi *embedded*, per motivi che verranno illustrati tra breve.

Lo standard internazionale ISO/IEC 9899 prevede, però, altri due modi di scrivere il "contenitore" appena analizzato. Il primo dei due è il seguente:

Listing 4.2: int return type with no parameters

```
int      main(void)
```

In questo caso il `main` ritorna un valore intero. Perché? A chi?

Questo primo tipo di dichiarazione di `main` viene usato in presenza di un sistema operativo o di uno *shell* che provvede a lanciare il suddetto programma. Siccome, però, è possibile che il programma contenuto nel `main` non porti a termine correttamente il proprio compito è importante ritornare al chiamante, cioè al sistema operativo, un codice di errore, che indichi se il programma è terminato correttamente (ciò viene indicato solitamente con il codice d'errore 0) oppure no, nel qual caso il programma ritorna un codice di errore, solitamente, negativo.

Il secondo modo previsto dallo standard ISO/IEC 9899 per dichiarare il `main` è dato di seguito:

Listing 4.3: int return type with parameters

```
int      main(int argc, char *argv[])
```

Anche questo secondo caso è da usarsi in presenza di un sistema operativo chiamante. Se viene utilizzata tale dichiarazione, i due parametri devono osservare determinare regole che, se osservate, permettono il passaggio di parametri dal sistema operativo al programma chiamato. Per maggiori dettagli sui due parametri e sulle regole che li governano si veda lo standard ISO/IEC 9899.

Nel prosieguo delle presenti pagine, salvo indicazione contraria, si ipotizzerà sempre la dichiarazione utilizzata dal codice 4.2.

Indipendentemente dal tipo di dichiarazione usato, dopo la dichiarazione (o meglio, dopo il *titolo*) sono presenti due parentesi graffe: la prima aperta e la seconda chiusa. Si veda a tal proposito il codice 4.1 nella pagina precedente.

Dette parentesi hanno il compito di confinare una struttura, in questo caso il programma vero e proprio. Ciò che è posto fra le suddette parentesi è detto

[1]Nell'ambiente di sviluppo di Arduino, il main è nascosto al programmatore, che accede solitamente, solo alle funzioni setup e loop.

corpo del main. Siccome nell'esempio 4.1 il programma è vuoto, nella prossima sezione si propone un primo programma effettivo.

4.2 Ciao mamma

In tutti i libri che si rispettino il primo programma si chiama sempre "*Hello world*". Questi appunti, che rispettabili non sono, si limitano ad un più modesto "*Ciao mamma*". Si veda il codice 4.4 a pagina 139:

Listing 4.4: Ciao mamma

```
#include <stdio.h>

int     main(void)
{
   printf(``Ciao mamma\n'');
   return 0;    //Esce senza errore
}
```

Ci sono alcune novità nel codice. La prima è data proprio dalla prima riga, alla quale conviene dedicare una sezione apposita.

4.2.1 La direttiva di inclusione

La prima riga si apre in maniera piuttosto criptica: con il simbolo #. Tale simbolo annuncia la presenza di una *direttiva del preprocessore*. Per primo va definito il termine *preprocessore*:

Definizione 6 (Preprocessore)**.**
Il preprocessore è un software applicativo che esegue delle operazioni sul testo di un codice sorgente. Tale applicazione viene eseguita prima della compilazione del codice sorgente.

Le operazioni citate possono essere delle inclusioni di testo, delle sostituzioni di testo, delle definizioni di equivalenza e così via. Le più importanti verranno introdotte man mano che si renderà utile.

Definizione 7 (Direttiva)**.**
La direttiva definisce l'operazione che il preprocessore deve eseguire sul codice sorgente.

Fra queste, una delle più importanti è proprio la direttiva di inclusione. Essa permette di *includere* (aggiungere) una libreria al programma sorgente. La prima riga del codice 4.4 include la libreria standard `stdio.h`.

L'identificatore `stdio` indica la contrazione di *standard input output* e rappresenta la libreria contenente le funzioni di I/O (*input/output*) del linguaggio C, ovvero quelle funzioni che gestiscono l'*input* dei dati (da tastiera, da linea seriale, da file, ecc.) e l'*output* (su schermo, file, stampante, ecc.). In particolare è necessario includere la libreria `stdio` a causa della presenza nel `main` della funzione `printf`, definita proprio in tale libreria.

Il suffisso `.h` indica che il file in questione è di tipo *header*, ossia di intestazione. L'argomento *file di intestazione* e *file di implementazione* verrà trattato nei prossimi capitoli.

Le direttive devono osservare alcune semplici regole:

- le direttive devono inziare con il simbolo #
- le direttive non necessitano di terminatore (;)
- ogni riga non deve ospitare più di una direttiva

Rimane da spiegare il significato delle parentesi angolari che incorniciano il nome del file da includere. Un ipotetico file `libreria` può essere incluso nei tre seguenti modi:

- `#include <libreria.h>`
- `#include "libreria.h"`
- `#include "C://cartella1/cartella2/libreria.h"`

Nel primo caso sono utilizzate le parentesi angolari. Se si usa questa notazione sintattica la libreria da includere deve trovarsi in una *directory* nota all'ambiente di sviluppo ed appositamente dedicata a ciò. Solitamente, di *default* tale cartella si chiama *include*. Se l'ambiente di sviluppo lo permette, possono essere aggiunte altre *directory*.

Nel secondo caso sono utilizzati i doppi apici e viene indicato solamente il nome del file e l'estensione. Se si usa questa notazione sintattica la libreria da includere deve risiedere nella stessa cartella del file sorgente che implementa il `main`.

Nell'ultimo caso deve essere fornito l'intero *path* di locazione del file. La libreria da includere può risiedere in una qualsiasi cartella.

4.2.2 Il corpo del *main*

Il corpo del `main` contiene due istruzioni: la `printf` e la `return`. Mediante la prima istruzione si visualizzano (talvolta si dice "si stampano") a video i parametri posti fra parentesi, ovvero si esegue un'operazione di *output*. Mediante la seconda istruzione si "ritorna" il valore 0, ovvero l'assenza di errori, al sistema operativo.

Entrambe le istruzioni verranno brevemente trattate.

4.2.2.1 Una breve introduzione alla *printf*

La `printf` è una funzione dichiarata nel file `stdio.h` ed *implementata* nel file `stdio.c`. I due predetti file formano la libreria `stdio`, contenente le funzioni di I/O del linguaggio C. A differenza di altri linguaggi, infatti, ad esempio il Pascal, le funzioni di I/O non sono *built-in* ovvero integrate nel linguaggio, ma sono esterne ad esso e aggiunte da libreria.

Il termine *implementare* è un neologismo importato dall'inglese *to implement* (rendere effettivo, attuare, eseguire). In gergo tecnico il file di *implementazione* è il file ove è scritto il codice sorgente delle funzioni e non solo la loro dichiarazione come nel file di *intestazione*.

Nella presente sezione viene introdotta brevemente la funzione di visualizzazione `printf`. Una trattazione più approfondita verrà effettuata nel prossimo capitolo. Per adesso è sufficiente conoscere alcune semplici regole per poter usare la funzione correttamente:

- è possibile visualizzare a video una **costante di tipo stringa**[2] nel seguente modo[3]:

  Listing 4.5: Stampa di costante di tipo stringa

  ```
  printf("Questa␣e'␣una␣costante␣di␣tipo␣stringa");
  ```

  Si noti che la stringa è racchiusa fra doppi apici, non fra coppie di apostrofi. Inoltre, si noti anche che l'accentazione della lettera "e" è effettuata mediante un apostrofo. Ciò si rende necessario perché gli elementi componenti le stringhe sono solamente quelli presenti nella tabella ASCII (vedi Appendice A) che, fra le lettere, contiene solo quelle dell'alfabeto inglese. Ciò implica anche che la suddetta stringa è formata da 38 caratteri e non da 37, come si potrebbe ipotizzare ad una prima analisi.

  Qualche attento studente avrà notato una importante differenza fra la stringa costante del codice 4.5 e quella del codice 4.4 a pagina 139: la seconda stringa termina con `\n`. La differenza fra la prima scrittura e la seconda è la seguente: nel primo caso il *prompt* lampeggia alla fine della stringa e se si effettua una seconda stampa, questa verrà eseguita a fianco della prima; nel secondo caso il *prompt* lampeggia a capo della riga seguente e una eventuale seconda stampa verrà effettuata in tale posizione.

  Infine, si ponga attenzione al fatto che non si può stampare la costante numerica `1234` passando detto valore come parametro alla `printf`, ma si deve passare la stringa `"1234"`.

- è possibile visualizzare a video una **variabile** nel seguente modo:

  Listing 4.6: Stampa di variabile

  ```
  printf("%d", pippo);
  ```

  La variabile in questione è `pippo`, che supponiamo ancora essere di tipo `int`. Non è però sufficiente, come si vede dal codice 4.6, indicare la sola variabile da stampare: si deve anche indicare il **formato** di stampa. Per ogni variabile da stampare va quindi passata alla funzione una **coppia** di parametri separati da virgola: il formato e la variabile. Il formato va specificato sotto forma di stringa, che deve precedere la variabile. Nel presente caso il formato numerico intero è specificato dalla notazione `"%d"`. Tale notazione si presta alla seguente sintassi:

  Listing 4.7: Esempio di stampa di variabile

  ```
  printf("Il␣valore␣%d␣e'␣quello␣assunto␣da␣pippo", pippo);
  ```

  Se `pippo` assume il valore 7, la frase che verrà stampata sarà: "Il valore 7 e' quello assunto da pippo".

- è possibile stampare una **serie di variabili** nel seguente modo:

[2] Per il momento è sufficiente definire la stringa come un testo alfanumerico.

[3] L'evidenziazione degli spazi fra parola e parola è una scelta tipografica di LaTeX, ossia del programma usato per redarre i presenti appunti.

Listing 4.8: Serie di variabili

```
int val1;
char val2;

val1 = 1234;
val2 = 'K';
printf("Valore␣1␣=␣%d␣e␣valore␣2␣=␣%s.", val1, val2);
```

La frase stampata è la seguente: “Valore 1 = 1234 e valore 2 = K.”.

Per il momento queste poche nozioni sulla funzione `printf` possono essere sufficienti e ci permetteranno di affrontare gli esercizi che verrranno proposti fra poco.

4.2.2.2 L’istruzione *return*

La seconda istruzione che compare nel corpo del `main` è la `return`. Detta istruzione permette di abbandonare la funzione per tornare al chiamante. Il chiamante può essere a sua volta una funzione, oppure, come nel presente caso, il sistema operativo. Quindi nel caso del codice 4.4 a pagina 139 l’istruzione `return` permette l’uscita dal programma ed il ritorno al sistema operativo.

L’istruzione di ritorno deve però osservare alcune semplici regole. Si faccia riferimento, a tal proposito, alla fig. 4.1.

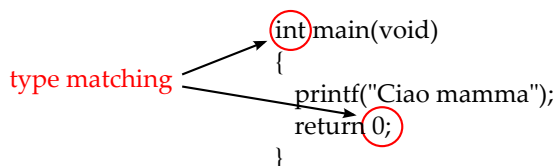



Figura 4.1: *Return-type rule*

La prima regola è quella del *type matching*: il valore che segue la parola chiave `return` deve essere dello stesso tipo di quello ritornato dalla funzione[4]. Tale valore può essere dato da una costante, oppure una variabile, una funzione o un’espressione. Si noti che ciò implica che se la funzione ha un `return-type` di tipo `void`, ossia che non ritorna nulla, anche il `return` non deve ritornare alcunché. In tal caso la sintassi richiede che si scriva il `return` seguito dal solo punto e virgola e non dal `void`. Se il `return` senza valori è posto come ultima riga della funzione, si può omettere.

La seconda regola è una convenzione: non deve essere rigidamente osservata, ma *se* osservata rende il programma più leggibile. E’ consigliabile non porre troppi `return` in una funzione: così facendo, il programma rischia di diventare di difficile lettura. L’ideale sarebbe che in una funzione ci fosse solamente una istruzione di ritorno. Se, però, viene eseguito un `return` a metà funzione, si esce immediatamente dalla funzione senza eseguire la restante metà delle istruzioni.

Si potrebbe accennare ad una terza regola, ma si preferisce rimandare la trattazione ad altra occasione.

[4]Il compilatore accetta anche la semplice osservanza della correlazione già evidenziata in fig. 3.2 a pagina 126.

4.3 Qualche semplice esempio

E' arrivato il momento di scrivere qualche semplice programma. Si farà uso delle nozioni fin qui apprese, cercando di non dover aggiungere altra carne al fuoco. Pur non avendole trattate in maniera esaustiva, si daranno momentaneamente per scontate le quattro operazioni aritmetiche, integrando qualche nozione, soprattutto sulle divisioni intere e decimali, quando necessario.

I programmi presentati sotto forma di esercizio saranno doverosamente semplici e talvolta un po' "forzati", ma ciò è dovuto unicamente dal fatto di voler usare soltanto le poche nozioni apprese.

4.3.1 Celsius Vs. Fahrenheit

In alcune nazioni, come ad esempio gli Stati Uniti, la temperatura viene misurata in gradi Fahrenheit. Tale scala prende il nome dal fisico tedesco Gabriel Fahrenheit ed è caratterizzata da un punto di congelamento dell'acqua posto a 32 gradi Fahrenheit ed un punto di ebollizione dello stesso liquido di 212 gradi Fahrenheit.

Alla luce di quanto detto si propone il seguente

Esercizio - ◊◊◊ Conversione Celsius-Fahrenheit
Si tracci uno specchietto che correli le seguenti temperature espresse in gradi Celsius in altrettante espresse in gradi Fahrenheit: +10°C, +20°C, +30°C, +40°C, +50°C.

Soluzione L'intervallo di temperature che vanno dal punto di congelamento all'ebollizione dell'acqua è suddiviso in 100 gradi nella scala Celsius e in 180 gradi nella scala Fahrenheit. Ciò significa che è possibile disegnare il seguente grafico Celsius/Fahrenheit:

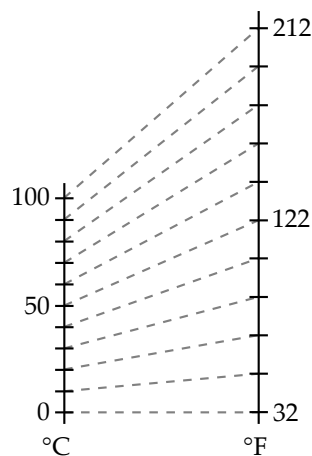



Figura 4.2: Corrispondenza Celsius/Fahrenheit

L'ampiezza della scala Fahrenheit visualizzata in fig. 4.2 è 1.8 volte maggiore. Questa osservazione ci porta a formulare la seguente relazione:

$$F = C \cdot 1.8 + 32 \tag{4.1}$$

e a ottenere i seguenti valori di equivalenza:

$$+10°C = +50°F$$
$$+20°C = +68°F$$
$$+30°C = +86°F$$
$$+40°C = +104°F$$
$$+50°C = +122°F$$

Il passo successivo è il diagramma di flusso relativo a come si intende sviluppare la soluzione[5]:

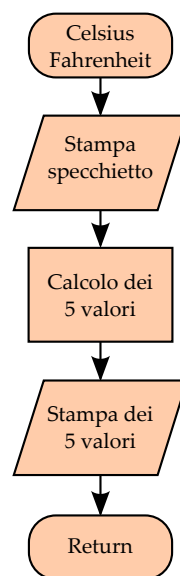



Figura 4.3: Diagramma Celsius/Fahrenheit

Il primo blocco provvede alla stampa dello specchietto ovvero alla stampa della prima riga che riporta la scritta "Celsius $\rightarrow$ Fahrenheit".

Il secondo blocco esegue i calcoli relativi ai 5 valori Fahrenheit, mentre il terzo blocco provvede alla stampa in colonna dei 5 valori. Per semplicità ogni riga corrisponde ad una stampa separata.

Il programma è quindi il seguente:

Listing 4.9: Celsius/Fahrenheit

```
int main(void)
{
   int f1, f2, f3, f4, f5;    // I 5 valori in gradi Fahrenheit
```

[5]E' facile obiettare che per problemi così semplici non serve tracciare il diagramma di flusso. Si è in disaccordo con tale tesi: lo studente salta sistematicamente lo *step* relativo al diagramma di flusso quando il problema è facile, salvo non essere poi capace di tracciarlo quando il problema è difficile. Per tale motivo si *ostenta* il diagramma di flusso anche e soprattutto quando l'esercizio è molto facile.

```
    // Stampa lo specchietto
    printf("Celsius␣␣␣->␣␣␣Fahrenheit\n");

    // Calcola i 5 valori. L'operatore di moltiplicazione e' il
    // simbolo *.
    f1 = 10*1.8+32;       // 10 gradi Celsius
    f2 = 20*1.8+32;       // 20 gradi Celsius
    f3 = 30*1.8+32;       // 30 gradi Celsius
    f4 = 40*1.8+32;       // 40 gradi Celsius
    f5 = 50*1.8+32;       // 50 gradi Celsius

    // Stampa i 5 valori, curando l'incolonnamento
    printf("␣␣10␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d\n", f1);
    printf("␣␣20␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d\n", f2);
    printf("␣␣30␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d\n", f3);
    printf("␣␣40␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d\n", f4);
    printf("␣␣50␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%d\n", f5);

    // Esce senza errore
    return 0;
}
```

Mediante le `printf` si è cercato di incolonnare grossolanamente i valori, dando all'insieme una forma simile a quella di uno specchietto. L'operatore di moltiplicazione, come indicato nei commenti, è il simbolo *.

Trattandosi del primo vero e proprio programma, lo studente farebbe bene a riflettere su cosa lo ha colpito in particolare. Le "prime impressioni" sono solitamente piuttosto importanti, perché in tali frangenti si ha la capacità di cogliere ciò che non è abituale e scontato per il proprio modo di pensare. Nei prossimi esercizi ciò non sarà più possibile.

Alcune osservazioni. I dati forniti dal problema erano più che sufficienti per poterlo affrontare e risolvere, senza la necessità di ricorrere a Internet. Non si vuole essere *demodé* o addirittura patetici, ma sovente ciò che fornisce risultati a breve termine allontana i risultati a lungo termine. La finalità di tutti gli esercizi che sono proposti nei presenti appunti mirano più a *formare* che a *risolvere*. Quindi complimenti a chi ha preso carta e penna e una tiratina d'orecchi a chi ha preso in mano il *mouse*. Due tirate d'orecchi a chi si è limitato a leggere!

Un'ulteriore riflessione è data dalla stesura del diagramma di flusso. Non si vuole apparire maniaci. E' evidente che un programmatore esperto di fronte ad un problema che ritiene molto semplice non farà mai un diagramma di flusso prima di scrivere il codice. Però si sottolinea che lo studente farebbe bene a coltivare tale pratica in modo da abituarsi a non risolvere i problemi in maniera impetuosa e incontrollata, ma seguendo sistematicamente un metodo. La stesura di un diagramma di flusso dà il tempo al programmatore di riflettere e valutare le soluzioni che affiorano alla mente.

Terza riflessione. I commenti precedono le istruzioni quasi sempre. Si è già avuto modo di illustrare l'importanza dei commenti: il commento obbliga alla riflessione e a delineare chiaramente i processi da eseguire. Molti studenti, per far felice l'insegnante aggiungono i commenti a programma finito e fun-

zionante. Non serve a nulla. Il commento è propedeutico all'istruzione, non il contrario.

Un'ultima riflessione. Si sono usati 5 tipi interi per memorizzare i relativi gradi espressi secondo la scala Fahrenheit. Ciò si è reso possibile per le particolari scelte di scala Celsius effettuate. Moltiplicare il coefficiente 1.8 per 10, 20, ecc. dà sempre un valore intero, per cui si è optato per la soluzione più semplice. Se si fosse reso necessario risparmiare memoria, si sarebbe potuto scegliere anche il tipo `unsigned char` o semplicemente un `char`, che occupa solamente 8 bit, contro i 32 del tipo `int`. Uno specchietto più particolareggiato, invece, avrebbe sicuramente reso necessario l'uso di variabili di tipo `float`.

4.3.2 I numeri di Bernoulli

Jakob Bernoulli (1654 - 1705) fu un matematico svizzero e capostipite di una leggendaria famiglia di matematici. La sua opera principale, la *Ars Conjectandi*, venne pubblicata postuma nel 1713 e contiene i cosidetti numeri di Bernoulli, utilizzati nel calcolo approssimato del numero di Nepero e, secondo la seguente serie:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots \tag{4.2}$$

La notazione $n!$ si pronuncia "n fattoriale" oppure "fattoriale di n" e si calcola come

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \ldots \cdot n \tag{4.3}$$

e, per definizione, si ha che 0! = 1.

Tutto ciò suggerisce il seguente

Esercizio - ◊◊◊ Numeri di Bernoulli

Si calcoli il valore approssimato del numero di Nepero mediante i primi 8 termini della serie 4.2.

Soluzione

L'espressione 4.2 con 8 termini diventa quindi la seguente:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} \tag{4.4}$$

Il problema sembra piuttosto banale ed in effetti lo è. Ciò non significa però che la soluzione da adottare debba per forza essere la prima che viene in testa.

Si potrebbe pensare di svolgere il seguente calcolo:

$$e = 1 + 1 + \frac{1}{2} + \frac{1}{2 \cdot 3} + \frac{1}{2 \cdot 3 \cdot 4} + \frac{1}{2 \cdot 3 \cdot 4 \cdot 5} + \frac{1}{2 \cdot 3 \cdot 4 \cdot 5 \cdot 6} + \frac{1}{2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7} \tag{4.5}$$

che naturalmente è assolutamente corretto. Ha il solo difetto di complicarsi man mano che aumentano i termini.

Senza grande fatica si nota, però, che il denominatore aumenta il fattoriale di 1 ad ogni termine per cui esiste uno schema che si può cercare di sfruttare.

Quindi se un determinato termine ha denominatore $n!$, quello successivo avrà denominatore $n! \cdot (n+1)$.

Dando per scontati i primi due termini e focalizzando l'attenzione sul terzo termine si potrebbe ipotizzare

$$e \leftarrow 2$$
$$d \leftarrow 2$$
$$f \leftarrow 2$$

dove e rappresenta il risultato parziale, dopo due termini, della 4.4; d rappresenta il denominatore del terzo termine e f rappresenta l'ultimo fattore del fattoriale dopo tre termini.

Per calcolare il terzo termine della serie è sufficiente eseguire il seguente calcolo:

$$e = e + 1/d \tag{4.6}$$

e per calcolare i termini successivi basta eseguire

$$f \leftarrow f + 1$$
$$d \leftarrow d \cdot f$$
$$e \leftarrow e + 1/d$$

Ora che è stata sviluppata l'idea, si può tracciare il diagramma di flusso:

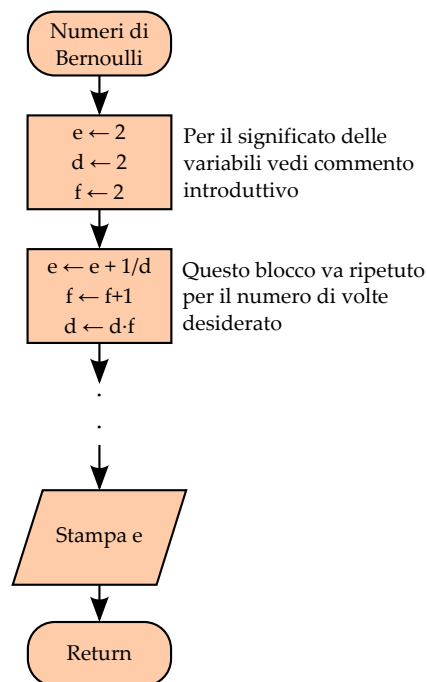



Figura 4.4: Diagramma Numeri di Bernoulli

Prima di scrivere il programma vero e proprio è opportuno fare ancora una riflessione sui tipi di dato da usare.

Siccome la variabile e rappresenta il numero di Nepero approssimato, è evidente che dovrà essere di tipo `float` o `double` a seconda della precisione

voluta. Si può supporre che, ai fini dell'esercizio, la precisione fornita dal tipo `float` possa essere sufficiente, senza ulteriori analisi.

Le variabili d ed f saranno sicuramente di tipo intero. Più precisamente, visto che f assume valori molto bassi (max. 7), potrebbe benissimo essere definita come tipo `short` o come `char`, mentre la variabile d dovrà per forza essere definita almeno come tipo `short`.

Rimane ancora da affrontare un problema piuttosto importante che, se non risolto, compromette totalmente il calcolo della serie. Di che problema si sta parlando?

◊◊◊

L'assegnazione sottostante ha due termini al secondo membro.

$$e \leftarrow e + 1/d$$

Il primo termine è di tipo `float`, mentre il secondo è un rapporto fra interi ($1/d$) e pertanto verrà eseguito mediante divisione intera. Naturalmente, essendo il denominatore sempre maggiore del numeratore, il quoziente di tale divisione sarà sempre 0. Per risolvere il problema si deve modificare il tipo della variabile d, utilizzando ad esempio un `float` oppure modificare l'espressione nel seguente modo:

$$e \leftarrow e + 1.0/d \tag{4.7}$$

In tal modo il numeratore della frazione non è più di tipo intero e la divisione verrà effettuata in modo decimale. A questo punto è possibile scrivere il codice del programma:

Listing 4.10: Numeri di Bernoulli

```
int main(void)
{
   float e;      // Risultato parziale
   short d;      // Denominatore del termine
   char f;       //Ultimo fattore del fattoriale

   d = 2;        //Inizializzazione delle variabili
   f = 2;
   e = 2;

   e = e+1.0/d; //Terzo termine
   f = f+1;
   d = d*f;

   e = e+1.0/d; //Quarto termine
   f = f+1;
   d = d*f;

   e = e+1.0/d; //Quinto termine
   f = f+1;
   d = d*f;

   e = e+1.0/d; //Sesto termine
```

```
    f = f+1;
    d = d*f;

    e = e+1.0/d; //Settimo termine
    f = f+1;
    d = d*f;

    e = e+1.0/d; //Ottavo termine

    //Stampa il risultato ed esce senza errore
    printf("La serie di Bernoulli con 8 termini ");
    printf("produce il seguente risultato: %f", e);
    return 0;
}
```

L'istruzione di stampa è stata divisa in due righe per una mera questione di spazio. Si noti, però, che a video la frase appare su una sola riga.

Naturalmente il programma avrebbe potuto essere scritto in maniera molto più efficace utilizzando un ciclo. Si è optato per la presente soluzione per non dover anticipare l'argomento che verrà trattato nel capitolo sesto. Il fine dei presenti esercizi non è quello di produrre codice efficiente, ma di fornire spunti di riflessione sugli argomenti fin qui trattati.

4.4 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 4.

Capitolo 5

Le funzioni di I/O

Nel presente capitolo verranno affrontate specificatamente le due principali funzioni di *input/output*: la funzione di *output* `printf` e la funzione di *input* `scanf`.

La `printf` è già stata introdotta brevemente ed in maniera piuttosto informale. Nel presente capitolo verranno affrontati esaurientemente i diversi tipi di formato che la funzione di stampa supporta come pure i parametri di formattazione.

In maniera altrettanto dettagliata verrà studiata la funzione di *input* `scanf` e come utilizzarla con i diversi formati. Ciò ci permetterà di utilizzare la tastiera durante il *run-time* per eseguire l'*input* dei dati richiesti dal programma all'utente.

Non verranno affrontate nel presente capitolo le gestioni in *input* e *output* dei file, che saranno oggetto di capitolo a parte.

5.1 La funzione di *output*

Nella sezione 4.2.2.1 si è avuto modo di apprendere i rudimenti di visualizzazione a video mediante la funzione `printf`. La presente sezione tratterà l'argomento in maggior profondità.

Nella libreria `stdio` la funzione `printf` è dichiarata mediante il seguente *prototipo*[1]:

Listing 5.1: Prototipo printf

```
int printf(char *format, arg1, arg2, ...);
```

Dalla dichiarazione si apprende una cosa nuova: la funzione *restituisce un intero*. Si confrontino le due seguenti righe di codice:

[1] I concetti di *titolo* e *prototipo* verranno trattati nel capitolo 11.

Listing 5.2: Due differenti modi di usare printf

```
printf("Ciao␣mamma\n");          //Modo fin qui usato
cnt = printf("Ciao␣mamma\n");    //Modalita' conteggio carattere
```

La prima delle due righe illustra il modo fin qui usato della `printf`. Si nota che la funzione non ritorna alcun valore. Nel secondo caso la funzione ritorna un valore che viene assegnato alla variabile, che si suppone intera, `cnt`. Questi due casi riassumono la terza regola (non ancora illustrata) alla quale si è accenato nella sezione 4.2.2.1.

Questa è l'occasione buona per illustrarla. Se una funzione è dichiarata in modo tale che restituisca un valore, non è necessario che detto valore venga effettivamente "accolto" da una variabile: si può anche usare la funzione come se non restituisse nulla. Fino ad ora si è deciso di fare esattamente così. In realtà la `printf` *restituisce il numero di caratteri stampati*, quindi nel caso sopra illustrato si avrà `cnt` = 10. Solitamente, però, non è così importante conoscere il numero di caratteri stampati.

Il primo parametro della `printf` è la stringa di formattazione. Essa deve contenere tante formattazioni quanti sono i parametri che seguono, ossia `arg1`, `arg2`, `arg3`, ecc. Si vedano i seguenti esempi:

Listing 5.3: Esempi d'uso di printf

```
printf("%d%c%f", p1, p2, p3);   //Intero, carattere e float
printf("%d%d", p4, p5);         //Intero, intero
printf("%f%c%c", p6, p7, p8);   //Float, carattere e carattere
```

Nel primo caso si stampano tre variabili: la prima intera (ad esempio un `int`), la seconda di tipo carattere e la terza di tipo `float`.

Nel secondo caso si stampano due variabili: entrambe di tipo intero.

Nel terzo caso si stampano tre variabili: la prima di tipo `float`, la seconda e la terza di tipo carattere.

Non vi è limite al numero di parametri.

Le nozioni fin qui presentate potrebbero anche essere sufficienti per usare grossolanamente la funzione di stampa. Anche e soprattutto perché in fase di sviluppo di programmi commerciali non si usa la `printf` per eseguire l'*output*.

Nel caso, però, in cui la curiosità dovesse essere irrefrenabile e la sete di sapere irresistibile, di seguito vengono fornite le necessarie conoscenze sulla formattazione dell'*output*.

5.2 La formattazione dell'*output*

Per "formattazione dell'*output*" si intende quell'insieme di regole che permettono un determinato *layout* della stampa: allineamento a destra oppure a sinistra, numero di cifre dopo la virgola da stampare, ecc. Si tratta, quindi, di regole che permettono la parametrizzazione della stampa al fine di renderla più gradevole o meglio impaginata.

La sequenza "%*x*" contiene le *specifiche di conversione* e il carattere che segue il simbolo % è detto *carattere di conversione*. Si è già visto che i caratteri di

conversione possono essere diversi (fino ad ora ne sono stati presentati tre: *f* per i numeri in virgola mobile, *d* per i formati numerici interi e *c* per i caratteri) ma soprattutto esistono diverse varianti sintattiche.

Fra il simbolo % e il carattere di conversione, infatti, si può trovare, nell'ordine specificato:

- un **segno meno**. Se presente, specifica l'allineamento a sinistra del parametro;
- un **numero**. Se presente, l'argomento verrà stampato in un'ampiezza di campo almeno pari a quella indicata dal numero. Queste due regole sintattiche permettono diverse presentazioni del dato da stampare. Si vedano i sottostanti esempi:

Listing 5.4: Esempi di allineamento

```
int pippo;

pippo = 1234;
printf("%-8d\n", pippo); //Allineato a sinistra
printf("%8d\n", pippo);  //Allineato a destra
```

Le due stampe producono rispettivamente i seguenti allineamenti:

"1234␣␣␣␣"
"␣␣␣␣1234"

Entrambi i numeri vengono stampati occupando uno spazio complessivo di *almeno* 8 caratteri, ma la prima stampa provoca un allineamento a sinistra e la seconda un allineamento a destra.
Se il dato da stampare è superiore all'ampiezza di campo indicato non si ha allineamento. Si osservi il seguente esempio:

Listing 5.5: Ampiezza di campo insufficiente

```
int pippo;

pippo = 0123456789;
printf("%-8d\n", pippo); //Campo insufficiente
printf("%8d\n", pippo);  //Campo insufficiente
```

I due numeri non possono essere allineati a destra o a sinistra perché l'ampiezza di campo è insufficiente:

"0123456789"
"0123456789"

Si noti, infine, che se non viene specificato né il segno né l'ampiezza di campo (ad esempio solo `"%d"`) la stampa viene allineata a sinistra.
Quanto fin qui visto è valido anche per i caratteri, per le stringhe e per i numeri in virgola mobile, essendo i risultati assolutamente identici.
I numeri in virgola mobile permettono, però, anche la definizione della precisione, ovvero del numero di cifre dopo la virgola.
Esiste, però, anche la possibilità di definire la precisione del dato da stampare. Ciò ci introduce alle successive due regole;

- un **punto**. Se presente, ha il compito di separare l'ampiezza di campo dalla precisione;
- un **numero**. Se presente, identifica la precisione voluta del dato. Rimane da specificare cosa si intende, di volta in volta, per *precisione*.

La precisione assume tre diversi significati, a seconda che essa sia riferita a numeri in virgola mobile, a numeri interi o a stringhe. Più specificatamente la precisione è

- per un **numero in virgola mobile**: il numero di cifre dopo la virgola;
- per un **numero intero**: il numero minimo di cifre da visualizzare;
- per una **stringa**: il numero massimo di caratteri da visualizzare.

Alcuni esempi aggiungeranno chiarezza a quanto detto.

5.2.1 La precisione nella stampa dei numeri in virgola mobile

Si vedano le seguente righe di codice, tutte riferite alla stampa di un numero definito in virgola mobile:

Listing 5.6: Precisione di stampa dei numeri FP

```
float pluto;

pluto = 3.14;
printf("%-8.5f\n", pluto); //Allineamento a sinistra
printf("%8.5f\n", pluto);  //Allineamento a destra
```

Dette righe di codice producono il seguente effetto d stampa:

“3.14000␣”
“␣3.14000”

Si osserva che effettivamente i due numero appaiono uno allineato a sinistra e l'altro allineato a destra. Appare, però, subito evidente che i due numeri stampati differiscono dal numero assegnato a `pluto`, essendo stati aggiunti tre zeri non significativi in stampa. Ciò è dovuto al fatto che la precisione richiesta è stata indicata in 5 (`"%8.`**`5`**`f\n"`) e sono stati quindi aggiunti tanti zeri non significativi dopo la virgola fino al raggiungimento delle 5 cifre.

Inoltre, si nota pure che il numero complessivo di caratteri stampati è quello indicato dal primo dei due numeri definito nelle specifiche di conversione (`"%`**`8`**`.5f\n"`), ossia 8. Naturalmente, fra i caratteri conteggiati c'è anche il punto decimale.

Cosa succede se il numero cambia, ad esempio se il numero delle cifre dopo la virgola eccede la precisione richiesta? Si osservino le righe del codice 5.7:

Listing 5.7: Cifre superiori alla precisione

```
float pluto;

pluto = 3.141592;
printf("%-8.5f\n", pluto); //Allineamento a sinistra
printf("%8.5f\n", pluto);  //Allineamento a destra
```

In stampa si nota un troncamento del valore assegnato, dato che la cifra meno significativa non viene visualizzata:

"3.14159␣"
"␣3.14159"

In realtà in fase di stampa il valore non viene troncato ma *arrotondato*. Se il numero memorizzato in `pluto` fosse stato 3.141599 (chiedendo scusa al π) in fase di stampa sarebbe stato visualizzato il valore 3.14160.

Resta da chiarire cosa succede se alla variabile `pluto` viene assegnato il valore 123.141592, come nel codice 5.8:

Listing 5.8: Cifre superiori all'ampiezza di campo

```
float pluto;

pluto = 123.141592;
printf("%-8.5f\n", pluto); //Campo insufficiente?
printf("%8.5f\n", pluto);  //Campo insufficiente?
```

Succede esattamente quello che è successo nel codice 5.5: i due numeri non possono essere allineati a destra o a sinistra perché l'ampiezza di campo è insufficiente:

"123.14160"
"123.14160"

I numeri non vengono quindi mutilati in fase di stampa. Se il numero eccede l'ampiezza di campo, detta ampiezza viene semplicemente ignorata come pure l'allineamento.

5.2.2 La precisione nella stampa dei numeri interi

Nei numeri interi la precisione è più semplice da gestire. Nei numeri interi la precisione rappresenta semplicemente il numero di cifre *minimo* da visualizzare. Quindi se il numero da rappresentare ha un numero di cifre inferiore a quello minimo richiesto, vengono *aggiunti zeri non significativi* fino al raggiungimento della precisione richiesta.

Un esempio chiarirà ulteriormente il concetto.

Listing 5.9: Precisione di stampa dei numeri interi

```
int pippo;

pippo = 1234;
printf("%-8.5d\n", pippo); //Allineamento a sinistra
printf("%8.5d\n", pippo);  //Allineamento a destra
```

La precisione richiesta è 5, mentre il numero di cifre del numero è 4. Ciò significa che dovrà essere aggiunto uno zero non significativo in fase di stampa in modo da raggiungere la precisione richiesta:

"01234␣␣␣"
"␣␣␣01234"

Si noti che quanto detto sull'allineamento e sulla precisione nei numeri interi vale anche per le variabili di tipo `char`, laddove esse siano intese come rappresentative di numeri interi e non caratteri. Quindi un'ipotetica variabile

di tipo `char` alla quale dovesse venir attribuito il valore 12 verrebbe stampata nel seguente modo se le specifiche di conversione fossero rispettivamente `"%-8.5d\n"` e `"%-8.5d\n"`:

“00012␣␣␣”
“␣␣␣00012”

5.2.3 La precisione nella stampa delle stringhe

La precisione di stampa nelle stringhe utilizza altri criteri da quelli esposti per i numeri. Nelle stringhe la precisione di stampa indica il massimo numero di caratteri stampati. Si veda il codice 5.10:

Listing 5.10: Precisione di stampa delle stringhe

```
char p[] = "Ciao mamma";//Definizione e inizializzazione
                        //di stringa. Argomento trattato
                        //nel capitolo dedicato alle
                        //stringhe.

printf("%-9.7s\n", p); //Allineamento a sinistra
printf("%9.7s\n", p);  //Allineamento a destra
```

Il risultato delle due stampa è il seguente:

“Ciao ma␣␣”
“␣␣Ciao ma”

La stringa viene troncata (stavolta non viene arrotondata ☺) alla precisione richiesta, ovvero dopo 7 caratteri. E’ comunque buona pratica, quando si tronca una stringa, aggiungere dopo l’ultimo carattere dei puntini di sospensione in modo da evidenziare che la stringa è stata troncata, come nell’esempio seguente:

“␣␣Ciao ma...”

5.2.4 Gli specificatori di lunghezza

Davanti al carattere di conversione può essere posta una ulteriore lettera o una coppia di lettere. Questo ulteriore elemento è detto *specificatore di lunghezza* e ha il compito di definire con maggior chiarezza l’ampiezza della variabile, ovvero il numero di bit di cui è formata. Ciò si rende necessario in determinati casi per evitare errori di rappresentazione.

Per chiarire meglio il concetto si prende l’esempio della seguente specifica di conversione: `"%d"`. Essa può essere applicata a variabili intere di tipo `char` (8 bit), di tipo `short` (16 bit), tipo `int` (32 bit) o di tipo `long long int` (64 bit).

Una maggiore specificazione della lunghezza della variabile potrebbe essere utile nel caso di rappresentazioni in complemento a due. A ciò servono gli specificatori di lunghezza.

Lo standard ANSI C89 utilizzava solamente 2 specificatori di lunghezza, mentre lo standard ISO/IEC 9899 ne indica complessivamente addirittura 8.

Non si opta per nessuno dei due standard, ma si decide di trattare nella sottostante tabella un sottoinsieme degli specificatori C11, ritenuti più comunemente utilizzati.

Carattere	Effetto
`hh`	Se posto davanti a `d`, `i`, `o`, `u`, `x`, oppure `X` significa che la variabile è di tipo `signed char` oppure `unsigned char` e che il valore contenuto in essa va interpretato come un numero intero
`h`	Se posto davanti a `d`, `i`, `o`, `u`, `x`, oppure `X` significa che la variabile è di tipo `signed short` oppure `unsigned short`
`l` (elle)	Se posto davanti a `d`, `i`, `o`, `u`, `x`, oppure `X` significa che la variabile è di tipo `signed long int` oppure `unsigned long int`
`ll` (elle elle)	Se posto davanti a `d`, `i`, `o`, `u`, `x`, oppure `X` significa che la variabile è di tipo `signed long long int` oppure `unsigned long long int`
`L`	Se posto davanti a `f`, `e`, `E`, `g`, oppure `G` significa che la variabile è di tipo `long double`

Tabella 5.1: Specificatori di lunghezza

Il vecchio standard ANSI C89 prevedeva solamente gli specificatori di lunghezza `h` e `l`. In particolare quest'ultimo poteva essere usato anche nelle rappresentazioni in virgola mobile, per la stampa delle variabili di tipo `double`.

A causa delle diverse sovrapposizioni di standard, si possono ancora vedere notazioni simili a `printf("%lf", pluto);` dove la variabile `pluto` è definita di tipo `double`.

5.3 L'elenco dei caratteri di conversione

La trattazione delle specifiche di conversione dovrebbe continuare ancora per parecchie pagine, ma lo si ritiene tutto sommato inutile. L'argomento è vastissimo e molto dettagliatamente trattato nello standard ISO/IEC 9899, al quale si rimanda per approfondimenti personali.

Si è detto che si ritiene inutile l'approfondimento. Non si vorrebbe essere fraintesi a tal proposito. Una trattazione approfondita della formattazione di *output* mediante la `prinf` ha senso in programmi di tipo *Console* (ossia i vecchi programmi orientati al testo) e per niente adatti in ambiente grafico.

In realtà, lo studente che intraprende la scrittura di programmi *window oriented* (e come potrebbe fare diversamente?) dovrà apprendere metodi di formattazione completamente diversi e basati su altri concetti, molto lontani dai concetti illustrati nelle precedenti sezioni.

La libreria `stdio` potrebbe effettivamente tornare utile durante la scrittura su file. Ma anche in questo caso ci sarebbe da obiettare: la formattazione assu-

me senso se si "incolonnano" i dati, ad esempio, su uno *spreadsheet*. In tal caso, però, è decisamente più conveniente ricorrere a linguaggi quali SQL o simili.

Insomma, i motivi per cui si ritiene "inutile" approfondire ulteriormente l'argomento dovrebbero essere piuttosto chiari e motivati.

Si ritiene utile, a mo' di *reference*, aggiungere un parziale elenco dei caratteri di conversione più usati, indicati a suo tempo per l'ANSI C. Lo standard ISO/IEC 9899 ha notevolmente allargato l'argomento nel 2011, per cui, per i motivi già descritti, si ritiene sufficiente la vecchia tabella ANSI del 1989.

Carattere	Stampato come ...
`d`, `i`	`int`; numero intero
`o`	`int`; numero ottale senza segno (senza zero inziale)
`x`, `X`	`int`; numero esadecimale senza segno (senza `0x` o `0X` iniziale)
`u`	`int`; numero intero senza segno
`c`	`char`; carattere singolo
`s`	`char *`; stringa fino terminatore `\0` o al raggiungimento della precisione
`f`	`double`; in formato `[-]m.dddddd` con numero di cifre dopo la virgola indicato dalla precisione (*default* 6)
`e`, `E`	`double`; in formato `[-]m.dddddde±xx` oppure `[-]m.ddddddE±xx` con numero di cifre dopo la virgola indicato dalla precisione (*default* 6)
`g`, `G`	`double`; usa `%e` oppure `%E` se l'esponente è minore di -4 o maggiore o uguale alla precisione; altrimenti usa `%f`. Gli zeri non significativi non vengono stampati
`p`	`void`; puntatore
`%`	stampa un `%`

Tabella 5.2: Caratteri di conversione printf (ANSI C89)

5.4 La funzione di *input*

La funzione di *input* `scanf` permette l'immissione di dati da tastiera e l'assegnazione di detti dati ad una determinata variabile. Mediante la suddetta funzione sarà possibile attribuire valori digitati da tastiera a varibili di tipo `char`, `int`, `float`, `double` e alle variabili di tipo stringa.

Nella libreria `stdio` la funzione `scanf` è dichiarata nel seguente modo:

Listing 5.11: Prototipo scanf

```
int scanf(char *format, arg1, arg2, ...);
```

La funzione di *input* legge i caratteri digitati da tastiera (o meglio, dallo *standard input*) secondo il formato indicato da `char *format` e li attribuisce alle variabili indicate dagli argomenti che seguono. Si noti che ciascuno di detti argomenti deve avere la forma di un *puntatore*[2].

[2] Argomento, importantissimo, che verrà trattato nel capitolo 10. Per il momento basti sapere che il puntatore indica con precisione l'area di memoria ove la variabile è allocata.

Alcuni semplici esempi di uso della `scanf` sono i seguenti:

Listing 5.12: Esempi di scanf

```
char dotto;
int brontolo;
float pisolo;
double eolo;
char gongolo[20];        //Definizione di stringa.
                         //E siccome hanno tanto
                         //insistito, si citano anche
                         //mammolo e cucciolo.

scanf("%c", &dotto);
scanf("%d", &brontolo);
scanf("%f", &pisolo);
scanf("%lf", &eolo);     //Si noti la notazione lf
scanf("%s", gongolo);    //Si noti la mancanza di &
```

Le osservazioni da fare sono *molte e grandi* (cit.).

Le prime quattro istruzioni contengono l'*operatore di indirizzamento* `&`. Esso è anteposto alla variabile ed ha il compito di estrarre l'indirizzo di memoria ove la variabile è allocata e passarlo alla `scanf`. Tale indirizzo di memoria è detto *puntatore*.

La definizione di stringa è espressa come `char gongolo[20]`. La sintassi verrà trattata a tempo debito quando si parlerà di stringhe. Per il momento è sufficiente sapere che detta definizione alloca 20 caratteri in memoria, puntati dall'identificatore `gongolo`. Si anticipa, quindi, che `gongolo` non è una variabile di tipo `char`, ma un *puntatore* a `char`. Ciò spiega l'assenza dell'operatore di indirizzamento.

La prima istruzione prevede l'immissione di un carattere da tastiera. Il carattere può essere un qualsiasi carattere della tabella ASCII (vedi Appendice A), sia che esso sia *printable* oppure no. L'immissione si deve concludere con la digitazione del *carriage return* (Invio).

La seconda istruzione implementa l'immissione di un numero intero. Non vi è alcun controllo sul fatto che *effettivamente* venga digitato un numero intero. Semplicemente quanto digitato viene interpretato come 32 bit in complemento a due, senza alcun tipo di controllo. L'immissione si deve concludere con la digitazione del *carriage return*.

La terza e la quarta istruzione permettono l'immissione di numeri in virgola mobile. Si deve far attenzione ad usare il punto decimale e non la virgola come separatore della parte intera e la parte decimale. L'immissione si deve concludere con la digitazione dell'Invio.

L'ultima istruzione permette l'immissione di una stringa. Si noti la mancanza dell'operatore di indirizzamento, dato che, come si vedrà nel capitolo 9, il nome della stringa è già puntatore a se stessa. L'immissione si deve concludere con la digitazione dell'Invio.

La prossima sezione tratterà un piccolo esempio su come usare la `scanf`. Non si ritiene necessario approfondire ulteriormente l'argomento per gli stessi

motivi già esposti parlando della funzione `printf`. Non è necessario trattare nemmeno la formattazione dell'*input*, essendo essa praticamente uguale a quella vista per l'*output*.

5.4.1 Un esempio

L'uso delle due funzioni di I/O e la possibilità di acquisire e rappresentare dei dati formattati permette di "impaginare" con maggior grazia i risultati dei problemi dati. Il prossimo esercizio mira proprio a curare un po' di più tale aspetto, senza tuttavia cadere nell'eccesso opposto.

Esercizio ◊◊◊ Il giorno di nascita

Si chiede l'immissione della propria data di nascita **sulla quale non si chiede di effettuare alcun controllo né sintattico né semantico** (che si suppone quindi immessa correttamente) e il calcolo, con conseguente stampa, del relativo giorno della settimana. Quest'ultimo è rappresentato da un numero compreso fra 0 e 6, estremi compresi, secondo il seguente specchietto:

Valore	Giorno
0	Domenica
1	Lunedì
2	Martedì
3	Mercoledì
4	Giovedì
5	Venerdì
6	Sabato

Tabella 5.3: Equivalenza valore-giorno della settimana

Si chiede la stampa di detto specchietto in modo da agevolare l'utente nella interpretazione del risultato.

I passi da seguire per calcolare il giorno della settimana partendo da una data valida sono (il simbolo * indica moltiplicazione; il simbolo / indica divisione intera, trascurando il resto; il simbolo % indica il resto della divisione intera):

- Prepara calcolo dell'anno: a ← (14–mese)/12
- Calcola l'anno: y ← anno–a
- Calcola il mese: m ← mese+12*a–2
- Calcola il giorno: d ← (giorno+y+y/4–y/100+y/400+(31*m)/12)%7

La variabile `d` assumerà uno dei 7 valori indicati nella tabella 5.3. Si curi quanto più possibile il *layout* di quanto appare a video.

Soluzione

Appare abbastanza evidente che la soluzione dell'esercizio non implica alcuna profonda analisi, trattandosi di un problema piuttosto semplice. La difficoltà consiste nel corretto uso delle formattazioni, necessarie se si vuole presentare un *layout* gradevole.

D'altronde la difficoltà degli esercizi non può essere ancora aumentata, essendo lo studente sprovvisto di strutture fondamentali per poter risolvere problemi più complessi.

Comunque, indipendentemente dalla difficoltà presentata, il primo passo da fare consiste nella definizione del diagramma di flusso. Sostanzialmente possono essere immaginati quattro blocchi fondamentali: un blocco di presentazione del problema e successivo blocco di *input* dei dati; un terzo blocco di applicazione dell'algoritmo di calcolo; un blocco finale in cui si visualizza lo specchietto di tabella 5.3 ed il risultato calcolato dall'algoritmo.

Un esempio di un siffatto diagramma potrebbe essere quello presentato in fig. 5.1.

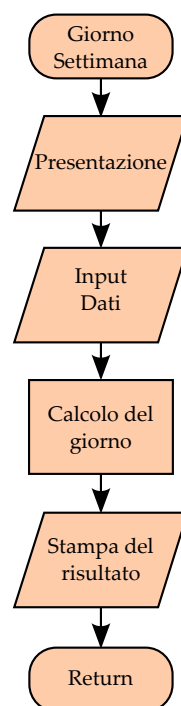



Figura 5.1: Diagramma Giorno della Settimana

Il primo blocco è quello di presentazione. E' cortese scrivere due righe che introducano l'utente su cosa sta per succedere. Probabilmente gli verrà chiesto di digitare qualcosa sulla tastiera, ad esempio la propria data di nascita, e l'ignaro utente si chiederà perché dovrebbe farlo. Le suddette due righe dovrebbero informarlo e tranquillizzarlo.

Un esempio di ciò potrebbe essere il codice seguente:

Listing 5.13: GdS 0v10

```
#include <stdio.h>       //Include la libreria standard di I/O

int main(void)
{
```

```
   // Visualizza due righe di introduzione del programma.
   printf("␣␣␣Il␣presente␣programma␣permette␣il␣calcolo\n");
   printf("del␣giorno␣␣della␣settimana␣a␣partire␣da␣una\n");
   printf("data␣digitata␣nel␣formato␣gg/mm/aaaa.\n");
   printf("Alcun␣controllo␣sulla␣correttezza␣della␣data\n");
   printf("viene␣eseguito.\n\n\n");
}
```

Il passo successivo è la richiesta di inserimento della data ed il relativo codice di *input*:

Listing 5.14: GdS 0v20

```
#include <stdio.h>      //Include la libreria standard di I/O

int main(void)
{
   // Variabili relative alla data da inserire
   int giorno;  // Data da inserire
   int mese;
   int anno;

   // Visualizza due righe di introduzione del programma.
   printf("␣␣␣Il␣presente␣programma␣permette␣il␣calcolo\n");
   printf("del␣giorno␣␣della␣settimana␣a␣partire␣da␣una\n");
   printf("data␣digitata␣nel␣formato␣gg/mm/aaaa.\n");
   printf("Alcun␣controllo␣sulla␣correttezza␣della␣data\n");
   printf("viene␣eseguito.\n\n\n");

   // Chiede all'utente di inserire una data valida nel
   // formato gg/mm/aaa.
   printf("Si␣digiti␣una␣data␣nel␣formato␣gg/mm/aaaa:␣");
   scanf("%d/%d/%d", &giorno, &mese, &anno);
}
```

Si notino le specifiche di conversione della `scanf`: esse sono nel formato `"%d/%d/%d"` e non nel formato `"%d%d%d"` come si sarebbe potuto ipotizzare. La prima versione permette all'utente di digitare degli *slash* fra giorno e mese e fra mese e anno, rendendo l'*input* un po' più aggraziato.

Ora è possibile procedere all'esecuzione dell'algoritmo:

Listing 5.15: GdS 0v30

```
#include <stdio.h>      //Include la libreria standard di I/O

int main(void)
{
   // Variabili relative alla data da inserire
   int giorno;  // Data da inserire
   int mese;
   int anno;

   // Variabili relative al calcolo di anno (y),
   // mese (m) e giorno (d).
   int y;
   int m;
```

```
   int d;
   int a;       // Variabile temporanea

   // Visualizza due righe di introduzione del programma.
   printf("␣␣␣Il␣presente␣programma␣permette␣il␣calcolo\n");
   printf("del␣giorno␣␣della␣settimana␣a␣partire␣da␣una\n");
   printf("data␣digitata␣nel␣formato␣gg/mm/aaaa.\n");
   printf("Alcun␣controllo␣sulla␣correttezza␣della␣data\n");
   printf("viene␣eseguito.\n\n\n");

   // Chiede all'utente di inserire una data valida nel
   // formato gg/mm/aaa.
   printf("Si␣digiti␣una␣data␣nel␣formato␣gg/mm/aaaa:␣");
   scanf("%d/%d/%d", &giorno, &mese, &anno);

   // Il prossimo passo consiste nell'applicazione dello
   // algoritmo spiegato nel testo dell'esercizio. Si
   // inizia con il calcolo dell'anno:
   a = (14-mese)/12;
   y = anno-a;

   // Procede con il calcolo del mese:
   m = mese+12*a-2;

   //Ed infine e' possibile calcolare il giorno della settimana:
   d = (giorno+y+y/4-y/100+y/400+(31*m)/12)%7;
}
```

L'ultimo passo consiste nella stampa dello specchietto di tabella 5.3, al fine di rendere più semplice l'interpretazione del risultato da parte dell'utente, e la stampa del risultato vero e proprio.

Listing 5.16: GdS 1v00

```
#include <stdio.h>      //Include la libreria standard di I/O

int main(void)
{
   // Variabili relative alla data da inserire
   int giorno;  // Data da inserire
   int mese;
   int anno;

   // Variabili relative al calcolo di anno (y),
   // mese (m) e giorno (d).
   int y;
   int m;
   int d;
   int a;       // Variabile temporanea

   // Visualizza due righe di introduzione del programma.
   printf("␣␣␣Il␣presente␣programma␣permette␣il␣calcolo\n");
   printf("del␣giorno␣␣della␣settimana␣a␣partire␣da␣una\n");
   printf("data␣digitata␣nel␣formato␣gg/mm/aaaa.\n");
   printf("Alcun␣controllo␣sulla␣correttezza␣della␣data\n");
```

```
    printf("viene␣eseguito.\n\n\n");

    // Chiede all'utente di inserire una data valida nel
    // formato gg/mm/aaa.
    printf("Si␣digiti␣una␣data␣nel␣formato␣gg/mm/aaaa:␣");
    scanf("%d/%d/%d", &giorno, &mese, &anno);

    // Il prossimo passo consiste nell'applicazione dello
    // algoritmo spiegato nel testo dell'esercizio. Si
    // inizia con il calcolo dell'anno:
    a = (14-mese)/12;
    y = anno-a;

    // Procede con il calcolo del mese:
    m = mese+12*a-2;

    //Ed infine e' possibile calcolare il giorno della settimana:
    d = (giorno+y+y/4-y/100+y/400+(31*m)/12)%7;

    // Stampa lo specchietto
    printf("\n\n");
    printf("␣␣␣␣␣Valore␣␣␣␣␣Giorno\n");
    printf("␣␣␣␣␣--------------------\n");
    printf("␣␣␣␣␣␣␣␣0␣␣␣␣␣␣␣Domenica\n");
    printf("␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣Lunedi\n");
    printf("␣␣␣␣␣␣␣␣2␣␣␣␣␣␣␣Martedi\n");
    printf("␣␣␣␣␣␣␣␣3␣␣␣␣␣␣␣Mercoledi\n");
    printf("␣␣␣␣␣␣␣␣4␣␣␣␣␣␣␣Giovedi\n");
    printf("␣␣␣␣␣␣␣␣5␣␣␣␣␣␣␣Venerdi\n");
    printf("␣␣␣␣␣␣␣␣6␣␣␣␣␣␣␣Sabato\n\n\n");

    // Stampa il risultato ed esce senza errore
    printf("Il␣giorno␣della␣settimana␣corrspondente␣alla\n");
    printf("data␣%.2d/%.2d/%d␣", giorno, mese, anno);
    printf("e'␣il␣seguente:␣%d␣(vedi␣specchietto)", d);

    return 0;
}
```

Si noti la specifica di conversione della data nella terzultima riga. La notazione `"data %.2d/%.2d/%d "` serve a stampare sia il giorno che il mese sempre con due cifre, anche nel caso che il giorno oppure il mese sia formato da una cifra solamente.

Naturalmente lo studente si deve fidare dell'algoritmo illustrato e probabilmente non ha capito a fondo la formulazione dello stesso. La finalità dell'esercizio, però, è altra, per cui non ci soffermeremo sulla struttura dell'algoritmo dato.

5.5 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 5.

Capitolo 6

Gli operatori aritmetico-logici

Gli operatori aritmetico-logici sono frequentemente fonte di grossi guai per lo studente, per cui è bene che il presente capitolo diventi oggetto di studio attento e puntiglioso. L'argomento è piuttosto infido e non perdona cali di attenzione o rilassamenti.

Contestualmente, si immagina anche che lo studente sia fremente e impaziente di scrivere programmi "veri", anche se semplici. Purtroppo si deve chiedere ancora un po' di pazienza: la strada è ancora lunga e qualche ostacolo deve essere ancora affrontato e superato. Si introdurrà qualche nuovo esercizio a partire dal prossimo capitolo.

6.1 Gli operatori aritmetici binari

Gli operatori aritmetici sono già stati praticamente introdotti nelle pagine precedenti. Inoltre, lo studente ha già una certa confidenza con essi, per cui le difficoltà sono ridotte al minimo.

L'elenco degli operatori aritmetici binari[1] è il seguente:

Operatore	Significato
+	somma
-	sottrazione
*	moltiplicazione
/	divisione intera/decimale
%	modulo

Tabella 6.1: Gli operatori aritmetici binari

[1] L'operatore binario si distingue da quello unario perché il primo ha due operandi, mentre il secondo ne ha uno solo.

L'operatore che attira maggiormente l'attenzione è sicuramente quello di divisione: lo stesso simbolo (/) rappresenta sia la divisione intera che quella decimale. Non è così per tutti i linguaggi. Ad esempio il Pascal utilizza simboli diversi per la divisione intera e per quella decimale.

Come è possibile valutare teoricamente la natura della divisione? Un primo sguardo al problema è stato dato in occasione dell'esercizio sui numeri di Bernoulli. Adesso, però, è arrivato il momento di approfondire leggermente l'analisi a suo tempo fatta. Si esamini il seguente codice:

Listing 6.1: Operatore di divisione

```
#include <stdio.h>

int main(void)
{
   int a = 15;
   int b = 2;
   float c = 3.0;
   float d;             // Risultato 1
   float e;             // Risultato 2

   // Le due operazioni danno lo stesso risultato?
   d = a/b/c;           // Operazione 1
   e = a/c/b;           // Operazione 2

   return 0;
}
```

Dal punto di vista strettamente algebrico le due operazioni 1 e 2 sono assolutamente identiche. Scrivere

$$x = a : b : c \tag{6.1}$$

e

$$y = a : c : b \tag{6.2}$$

è assolutamente la stessa cosa ed i due risultati, x e y, sono uguali. In Informatica le cose sono leggermente, ma sostanzialmente, diverse. Esaminiamo l'operazione 1.

Da sinistra verso destra la prima operazione è a/b. Si tratta di un'operazione di divisione fra interi, quindi il risultato sarà dato da un quoziente intero più un resto intero (eventualmente uguale a zero).

Nell'operazione 2, la prima divisione che viene eseguita è a/c. Il dividendo (o, se si preferisce, il numeratore) è di tipo intero, mentre il divisore è di tipo in virgola mobile in singola precisione. In questo caso l'operazione viene eseguita in virgola mobile in singola precisione, fornendo un risultato anch'esso in virgola mobile.

La differenza di cui si parlava innanzi appare ora evidente: l'Aritmetica non considera l'operazione di divisione un'operazione interna, mentre l'Informatica si. Ciò significa, secondo l'Aritmetica, che se si esegue la divisione 3:2 fra numeri relativi (insieme $\mathbb{Z}$) il risultato prodotto dalla divisione decimale apparterrà ai numeri razionali (insieme $\mathbb{Q}$), per cui non sarà appartenente all'insieme a cui appartengono gli operandi.

Nel campo dell'Informatica, invece, la stessa operazione viene eseguita fra variabili di tipo `int` ed il risultato sarà ancora di tipo `int` (operazione interna), ovvero appartenente allo stesso insieme degli operandi. Ciò implica differenze di valore fra i risultati dell'operazione 1 e 2.

6.1.1 Un errore frequente

Lo studente inciampa frequentemente in un errore piuttosto banale: la divisione per 0. Per l'Aritmetica una siffatta operazione è impossibile e ciò vale sostanzialmente anche per il linguaggio C[2].

Purtroppo, però, lo studente pone solitamente molta attenzione alla definizione del dominio di una funzione matematica[3], ma scarsa attenzione quando la stessa situazione si presenta in ambito informatico.

Invece bisognerebbe *sempre* assicurarsi che il valore di un divisore sia diverso da zero, a meno che non si sia sicuri che un tale valore sia impossibile in un contesto dato. In caso contrario, una divisione per zero produce un *crash* che interrompe bruscamente il programma, con effetti assolutamente spiacevoli per l'utente.

6.1.2 Associatività degli operatori aritmetici binari

Si è visto come il linguaggio C procede nell'individuazione del tipo di operazione di divisione (intera oppure decimale). Si è anche accennato, senza mai dichiararlo esplicitamente, al tipo di associatività di detto operatore. E' giunto il momento di esplicitare e generalizzare il concetto: tutti gli operatori della tabella 6.1 a pagina 167 hanno un'associatività che procede *da sinistra verso destra*. Ciò significa che nell'ipotetica espressione

$$x = a * b \% c / d + e - f; \tag{6.3}$$

il linguaggio C esegue prima l'operazione $a * b$; il risultato così ottenuto viene utilizzato per calcolare il modulo c seguito dalla divisione per d; il risultato così ottenuto viene prima sommato ad e ed infine si sottrae f.

6.1.3 Precedenza degli operatori aritmetici binari

Per quanto riguarda le precedenze, valgono quelle dell'aritmetica: prima vengono eseguite le operazioni di moltiplicazione, divisione e modulo e dopo quelle di somma e sottrazione.

Quindi la seguente espressione

$$x = a + b * c; \tag{6.4}$$

esegue prima l'operazione di moltiplicazione $b * c$ e poi la somma del risultato parziale della predetta moltiplicazione con a.

---

[2]In C++ o in altri linguaggi ad oggetti è possibile gestire l'eccezione di una divisione per zero, nel senso che si può decidere *cosa fare* se viene rilevato un divisore pari a zero. In ogni caso non viene comunque calcolato alcun valore.

[3]Gli insegnanti di matematica sono invitati a non esprimere commenti in proposito.

6.1.4 Le parentesi

Anche il linguaggio C, come l'Aritmetica, prevede l'uso delle parentesi nel caso in cui si voglia modificare la precedenza delle operazioni da svolgere. L'uso è assolutamente identico. Quindi le parentesi hanno precedenza assolutamente prioritaria. L'unica differenza consiste nei simboli usati e nei livelli di nidificazione. Iniziamo dai primi.

L'Aritmetica prevede tre simboli da usare nelle espressioni per modificare la precedenza degli operatori e quindi anche delle operazioni da svolgere:

- la parentesi graffa: "{ }";
- la parentesi quadra: "[ ]";
- la parentesi tonda: "( )".

Il linguaggio C riserva i primi due simboli a due significati distinti: Le parentesi graffe per racchiudere le strutture e le parentesi quadre per identificare gli elementi di un vettore[4]. Quindi solamente le parentesi tonde sono riservate per modificare le precedenze degli operatori.

Quello che può sembrare una limitazione, in realtà è un vantaggio: mediante i predetti tre tipi di parentesi si possono formulare tre livelli di nidificazione delle espressioni aritmetiche. Il linguaggio C permette, invece, un livello di nidificazione indefinito, ripetendo sempre lo stesso simbolo, come nell'esempio illustrato di seguito:

$$x = (a + b * (c + (d - e * (f - g\%(h + i)))))/j; \quad // \text{ 5 livelli di nidificazione}$$

6.2 Gli operatori aritmetici unari

Gli operatori aritmetici unari sono solamente due: l'operatore "+" e l'operatore "-". Esempi d'uso di detti operatori sono i seguenti:

$$x = (+a) + b * c; \tag{6.5}$$

$$y = a + (-b) * c; \tag{6.6}$$

Nell'espressione 6.5 la variabile a è preceduta dall'operatore unario "+": detto operatore lascia invariato il valore di a. Il che significa che se $a < 0$, l'espressione $(+a)$ sarà un valore negativo, mentre se $a > 0$, la stessa espressione sarà positiva.

Nell'espressione 6.6, invece, la variabile b è preceduta dall'operatore unario "-": detto operatore *nega* il valore della variabile b. Il che significa che se $b < 0$, l'espressione $(-b)$ sarà un valore positivo, mentre se $b > 0$, la stessa espressione sarà negativa.

6.2.1 Associatività degli operatori aritmetici unari

L'associatività degli operatori aritmetici unari è opposta a quella degli operatori aritmetici binari: *da destra verso sinistra*. Ciò è assolutamente ragionevole, dato che prima deve essere valutato il valore della variabile e poi, eventualmente, negato (se l'operatore è "-").

Non ci sono implicazioni degne di nota in una simile associatività.

[4]Argomento che verrà trattato nel capitolo 9.

6.2.2 Precedenza degli operatori aritmetici unari

Non ci sono differenze di precedenza fra i due operatori aritmetici unari. Tali differenze esistono, invece, se si considerano anche gli operatori aritmetici binari. Rispetto ad essi gli operatori unari hanno la precedenza su quelli binari. Una tabella che correli le precedenze di tutti gli operatori aritmetici è data di seguito:

Precedenza	Operatore
1	+ - (unario)
2	* / %
3	+ - (binario)

Tabella 6.2: Precedenza operatori aritmetici

Quindi, a mo' di riassunto, si può affrontare il seguente

Esercizio - ◊◊◊ La precedenza degli operatori
Sia data la sottostante espressione

$$x = a + b * (-c)\%d - (-(e + f * g))/h; \tag{6.7}$$

Si supponga che tutte le variabili siano intere ed assumano i seguenti valori:

$$a = 1; \quad b = 2; \quad c = 3; \quad d = 4; \quad e = 5; \quad f = 6; \quad g = 7; \quad h = 8;$$

Si chiede di calcolare il valore che assume la variabile x.

Soluzione
Naturalmente è possibile sostiuire i valori alle variabili, ottenendo in tal modo la seguente espressione:

$$x = 1 + 2 * (-3)\%4 - (-(5 + 6 * 7))/8; \tag{6.8}$$

Detta espressione verrà eseguita da sinistra verso destra, con le precedenze indicate in tab. 6.2, per cui i passaggi che verranno eseguiti dal compilatore saranno i seguenti:

$$\begin{aligned} x &= 1 + (-6)\%4 - (-(5 + 42))/8; \\ x &= 1 - 2 - (-47)/8; \\ x &= 1 - 2 + 5; \\ x &= 4; \end{aligned}$$

6.3 Gli operatori relazionali e di uguaglianza

Abbiamo imparato durante il nostro percorso scolastico che il risultato associato agli operatori aritmetici binari si chiama di volta in volta somma (+), differenza (-), prodotto (· oppure *, a seconda dell'ambito), quoziente (/) e resto (*mod* oppure %).

Il risultato associato agli operatori relazionali e di uguaglianza è, invece, detto *valore di verità* e può assumere solo due valori: *vero* (*true*) oppure *falso* (*false*). Alla luce di quanto verrà detto nelle prossime due sezioni, è di estrema importanza sottolineare come i valori di verità sono rappresentati, per cui si ritiene utile la seguente

Definizione 8 (Valori di verità)**.**
*Il valore di verità **false** è rappresentato dal valore 0 in formato* `int`*, mentre il valore di verità **true** è rappresentato dal valore 1 in formato* `int`*.*

Gli operatori relazionali sono i ben noti operatori studiati a scuola ed assumono lo stesso identico significato anche nel linguaggio C (anche se la simbologia è leggermente diversa):

Operatore	Significato	Esempio
>	maggiore	a > b
>=	maggiore o uguale	a >= b
<	minore	a < b
<=	minore o uguale	a <= b

Tabella 6.3: Gli operatori relazionali

Pure gli operatori di uguaglianza sono mutuati dall'Aritmetica e anche in questo caso la simbologia può sorprendere leggermente:

Operatore	Significato	Esempio
==	uguale	a == b
!=	diverso	a != b

Tabella 6.4: Gli operatori di uguaglianza

Si noti che se l'operatore è formato da due simboli, la sequenza (non interrotta da spazi) deve essere rispettata affinché l'operatore sia valido. Quindi si deve scrivere ">=" e non "=>", come pure si deve scrivere "!=" e non "=!".

6.3.1 Associatività degli operatori relazionali e d'uguaglianza

L'associatività di entrambe le categorie di operatori è *da sinistra verso destra*, come negli operatori aritmetici binari. Questa innocua affermazione, se non compresa a fondo può creare qualche dispiacere in sede di programmazione. Si veda il seguente

Esercizio - ◊◊◊ L'associatività degli operatori
Si suppongano le seguenti variabili intere e la sottostante espressione:

$$a = 5; \quad b = 3; \quad c = 1;$$

$$x = a < b < c; \tag{6.9}$$

La variabile intera x assume il valore di verità 1 oppure 0?

Soluzione

Probabilmente molti studenti avranno risposto che il valore di verità assunto da x sarà 0. Niente di più falso!

Infatti l'espressione 6.9 viene associata da sinistra verso destra, il che significa che la prima espressione che verrà calcolata sarà $a < b$, ossia $5 < 3$. Naturalmente l'espressione è falsa, *per cui sarà sostituita con il risultato 0*. L'operazione seguente diventa quindi $0 < c$ ovvero $0 < 1$, che è assolutamente vera, per cui il risultato di quest'ultima espressione sarà 1 e tale valore verrà attribuito alla variabile x.

6.3.2 Precedenza degli operatori relazionali e d'uguaglianza

Gli operatori relazionali hanno la precedenza sugli operatori di uguaglianza. Ad esempio, si veda la seguente espressione:

$$a < b == c < d \tag{6.10}$$

Prima vengono valutate le due espressioni relazionali ($a < b$ e $c < d$) e poi l'espressione di uguaglianza. Si supponga, ad esempio che

$$a = 1; \quad b = 2; \quad c = 3; \quad d = 4;$$

allora l'espressione 6.10 diventa

$$\begin{aligned} 1 < 2 == 3 < 4 \\ 1 == 1 \\ 1 \end{aligned} \tag{6.11}$$

La tabella riassuntiva delle precedenze fra gli operatori relazionali e quelli di uguaglianza è data di seguito:

Precedenza	Operatore
1	< <= > >=
2	== !=

Tabella 6.5: Precedenza degli operatori di uguaglianzae di uguaglianza

Si noti che gli operatori aritmetici hanno la precedenza sugli operatori relazionali. Ciò permette di non interpretare in maniera ambigua espressioni simili alla presente:

$$a < b + 1 \tag{6.12}$$

Detta espressione viene calcolata eseguendo prima l'operazione aritmetica $b + 1$ e poi il confronto con a. Il che significa che la 6.12 viene interpretata come $a < (b + 1)$.

6.3.3 Un altro errore frequente

Un errore che *tutti* i programmatori che utilizzano il linguaggo C hanno fatto almeno una volta consiste nell'usare l'operatore di assegnazione "=" come se fosse un operatore di uguaglianza "==". Si tratta di un errore infido e di non semplicissima localizzazione, una volta prodotto. Il concetto merita un semplice esempio.

Si suppongano le seguenti variabili e le relative assegnazioni:

$$a = 1; \quad b = 2;$$

L'espressione

$$a == b \tag{6.13}$$

produce evidentemente un valore di verità pari a 0, dato che l'uguaglianza non è rispettata. Se, però, il programmatore, per errore, scrive l'operatore di assegnazione al posto dell'operatore di uguaglianza, l'espressione diventa

$$a = b \tag{6.14}$$

che in tal modo assume il valore 2. Come si vedrà, però, nella prossima sezione, l'espressione 6.14 può essere considerata un'espressione relazionale semanticamente corretta. Questa affermazione contrasta, però, con la definizione 8 a pagina 172 e con il principio del terzo escluso (*Tertium non datur*). Una simile eventualità, naturalmente, non deve assolutamente avvenire.

Per scongiurare ciò, nella prossima sezione si vedrà come vi siano alcune strutture sintattiche che, per evitare contraddizione con il principio del terzo escluso, verifichino solamente se l'espressione relazionale produce risultato 0, *supponendo che sia 1 in caso contrario*.

Alla luce di ciò, l'espressione 6.14 assume il valore di verità 1, ossia esattamente l'opposto di ciò che si sarebbe ottenuto se l'espressione fosse stata la 6.13.

Trattandosi di un errore molto frequente, lo studente farebbe bene a memorizzarlo, per cercare di minimizzare la probabilità dell'infida svista.

6.4 Gli operatori logici

Nella sezione precedente si è parlato di *valori di verità*. Detti valori sono anche detti *valori booleani*, dal matematico britannico George Boole (1815 - 1864).

Boole pubblicò nel 1858 un libro intitolato "Indagine sulle leggi del pensiero su cui sono fondate le teorie matematiche della logica e della probabilità" (cfr. BOOLE [1]), nel quale tentò di formalizzare il pensiero logico, strutturandolo mediante leggi, teoremi e operazioni logiche. Il fine era quello di tradurre in forma algebrica la logica delle proposizioni.

Le teorie di Boole rimasero sconosciute ai più, fino a quando, nel 1938 presso il Massachusetts Institute of Thechnology, Claude Shannon non pubblicò a sua volta un testo intitolato "Un'analisi simbolica dei circuiti a relè a scatto" nel quale utilizzò abbondantemente il lavoro svolto dal matematico inglese.

Nel suo trattato Boole sviluppò un sistema che affondava le sue regole nel calcolo in base 2, anzichè nel più familiare sistema decimale.

Le *espressioni booleane* (ovvero quelle espressioni che possono avere solamente due valori di verità: *vero* oppure *falso*) utilizzano degli operatori logici, oltre a quelli relazionali e di uguaglianza visti nella precedente sezione, che sono ben noti dall'Algebra. I simboli $\vee$, $\wedge$ e $\neg$ utilizzati nella logica delle proposizioni hanno i loro equivalenti rispettivamente negli operatori booleani **OR**, **AND** e **NOT**.

Una tabella riassuntiva degli operatori logici è la seguente: Un'osserva-

Operatore	Simbolo Mat.	Significato	Esempio
&&	$\wedge$	AND logico	a && b
||	$\vee$	OR logico	a || b
!	$\neg$	NOT logico	!a

Tabella 6.6: Gli operatori logici

zione è dovuta riguardo all'operatore di negazione. Al fine di salvaguardare il principio del terzo escluso, l'operatore converte un qualsiasi operando il cui valore è *diverso da zero* in 0 (*false*) e in 1 (*true*) qualsiasi operando posto a 0.

6.4.1 Associatività degli operatori logici

Gli operatori logici, come quelli aritmetici binari hanno associatività *da sinistra verso destra*. Ciò non si rivela particolarmente importante dato che non vi è perdita di informazione nell'elaborazione delle espressioni booleane, trattandosi solamente di sviluppare dei calcoli relativi a semplici valori di verità.

Un altro aspetto può risultare, invece, piuttosto interessante durante la scrittura del codice. Si veda a tal proposito la seguente espressione:

$$a \mathbin{\&\&} b \mathbin{\&\&} c \mathbin{\&\&} d \mathbin{\&\&} e \tag{6.15}$$

E' evidente che è sufficiente che una sola variabile assuma il valore di verità *false*, che l'intera espressione assumerà quel valore, indipendentemente dallo stato delle restanti variabili.

Il linguaggio C *smette di elaborare ulteriormente l'espressione quando il valore di verità è già determinato*. Quindi se, nell'espressione 6.15 la variabile a assume il valore di verità *false*, l'espressione non viene ulteriormente valutata, ma si interrompe immediatamente. Tale tecnica è detta *short-circuit evaluation*, ovvero "valutazione di corto circuito".

Alla luce di ciò risulta evidente che conviene porre quelle espressioni facilmente valutabili a sinistra e quelle di calcolo più complesso a destra. Per semplicità, nei presenti appunti, si utilizzano come esempio spesso espressioni molto semplici, fatte da sole variabili. Si ricorda, però, che una qualsiasi espressione può essere sempre formata da costanti, variabili, funzioni o espressioni formate dagli elementi appena nominati.

Una funzione estremamente complessa e *time consuming* andrebbe quindi posta più a destra possibile nell'espressione.

6.4.2 Precedenza degli operatori logici

Le precedenze costituiscono spesso fonte di errori nella valutazione delle espressioni. Gli operatori logici non fanno eccezione. L'operatore di negazione "!" ha la stessa precedenza degli operatori aritmetici unari, mentre i restanti due operatori logici hanno precedenza molto bassa (la più bassa fin qui vista) e comunque posta sotto gli operatori di uguaglianza. All'interno di dette precedenze, l'operatore && ha la precedenza sull'operatore ||.

E' quindi fondamentale porre molta attenzione ad espressioni simili alla seguente:

$$x = a < b + c \,||\, d \;\&\&\; e \ * f - g; \tag{6.16}$$

Per semplificare l'esempio si suppongano i seguenti valori attribuiti alle variabili:

$$a = 1; \quad b = 2; \quad c = 3; \quad d = 4; \quad e = 5; \quad f = 6; \quad g = 7;$$

L'espressione 6.16 diventa quindi la seguente:

$$x = 1 < 2 + 3 \,||\, 4 \;\&\&\; 5 \ * 6 - 7;$$

Prima devono essere elaborate le espressioni aritmetiche, ad iniziare dalle moltiplicazioni, poi quelle relazionali ed infine quelle logiche ad iniziare dalle AND logiche. Quindi:

$$\begin{aligned} &x = 1 < 5 \,||\, 4 \;\&\&\; 30 - 7; \qquad (6.17)\\ &x = 1 \,||\, 4 \;\&\&\; 24;\\ &x = 1 \,||\, 1;\\ &x = true; \end{aligned}$$

6.5 Gli operatori di incremento e decremento

Gli operatori di incremento e decremento sono complessivamente quattro e sono rappresentati nella tabella 6.7:

Operatore	Nome	Esempio
++	postincremento	a++
++	preincremento	++a
- -	postdecremento	a- -
- -	predecremento	- -a

Tabella 6.7: Gli operatori di incremento/decremento

Si nota dallo specchietto che gli operatori di postincremento e di preincremento si distinguono non attraverso il simbolo usato, che è sempre lo stesso (++), ma attraverso la posizione che occupa rispetto alla variabile ($a++$ piuttosto che $++a$). Analogo ragionamento si può fare per gli operatori di postdecremento e predecremento.

Con le sole argomentazioni fornite fin qui è assolutamente impossibile comprendere quale sia la differenza fra postincremento e preincremento oppure fra postdecremento e predecremento.

Come al solito si ricorre ad un esempio che si spera sia illuminante. Si veda il codice 6.2 a pagina 177:

Listing 6.2: Uso dell'operatore di postincremento

```
int pippo;
int a=5;

pippo = a++;            //Assegnazione con postincremento
printf("pippo␣=␣%d\n", pippo);  //Quanto vale pippo?
printf("a␣=␣%d\n", a);          //Quanto vale a?
```

Dopo l'esecuzione dell'istruzione di assegnazione con postincremento viene stampato il valore di `pippo`. Tale valore srà pari a 5, quindi `pippo = 5`. Quando viene stampato il valore di `a`, invece, si ha `a = 6`. Quindi possiamo avanzare la seguente

Definizione 9 (Postincremento)**.**
*L'operatore di postincremento esegue l'incremento della variabile alla quale è abbinato **dopo** che l'espressione*[5] *associata è stata eseguita.*

Ciò significa che l'assegnazione con postincremento del codice 6.2 è equivalente al seguente:

Listing 6.3: Assegnazione equivalente al postincremento

```
pippo = a;      //Assegnazione ...
a = a+1;        //... equivalente
```

A questo punto, diventa anche più comprensibile la seguente

Definizione 10 (Preincremento)**.**
*L'operatore di postincremento esegue l'incremento della variabile alla quale è abbinato **prima** che l'espressione associata venga eseguita.*

Quindi il seguente codice

Listing 6.4: Uso dell'operatore di preincremento

```
int pippo;
int a=5;

pippo = ++a;            //Assegnazione con preincremento
printf("pippo␣=␣%d\n", pippo);  //Quanto vale pippo?
printf("a␣=␣%d\n", a);          //Quanto vale a?
```

è equivalente al seguente

Listing 6.5: Assegnazione equivalente al preincremento

```
a = a+1;        //Assegnazione ...
pippo = a;      //... equivalente
```

[5] Si noti che è stata usato il termine "espressione" e non "istruzione". La distinzione è assolutamente fondamentale. Se usassimo il termine "istruzione" dovremmo distinguere un caso in cui la definizione sarebbe errata. Detto caso verrà esaminato nel capitolo 8.

Quindi la stampa del codice 6.4 presenta gli stessi valori sia per `pippo` che per `a`, ossia 6.

Alla luce di quanto detto non si ritiene necessario fornire ulteriori due definizioni associate agli operatori di decremento. Si ritiene, invece, utile sottolineare l'importanza di quanto esposto nella presente sezione anche in virtù del fatto che la semplicità dell'argomento è solo apparente.

6.5.1 Associatività degli operatori di incremento/decremento

L'associatività degli operatori di incremento/decremento è diversa a seconda del fatto che l'operatore sia *prefisso* (es. $++a$ oppure $--a$) oppure *postfisso* (es. $a++$ oppure $a--$).

Ciò è piuttosto logico, trattandosi di operatori unari: vi è la necessità di valutare *prima* la variabile e *poi* l'effetto introdotto dall'operatore. La tabella seguente riassume il tipo di associatività degli operatori di incremento e decremento:

Operatore	Associatività
postincremento	da sinistra verso destra
postdecremento	da sinistra verso destra
preincremento	da destra verso sinistra
predecremento	da destra verso sinistra

Tabella 6.8: Associatività degli operatori di incremento/decremento

Anche in questo caso vi sono degli aspetti che vanno analizzati con un po' di attenzione. Per poterlo fare in maniera rigorosa è bene fornire la seguente

Definizione 11 (Punto di sequenza)**.**
Per ***punto di sequenza*** *(sequence point) si intende un determinato passo della sequenza di calcolo di un'espressione aritmetico-logica.*

Tenedo presente la suddetta definizione, lo standard ISO/IEC 9899 dice che

> *"fra un punto di sequenza e quello successivo un determinato oggetto*[6] *non dovrebbe essere modificato nel proprio valore più di una volta durante la valutazione dell'espressione."*

Si veda, però, la sottostante espressione di assegnazione con preincremento:

$$i = ++i + 1; \tag{6.18}$$

Questa espressione modifica *due volte* il valore della stessa variabile durante la valutazione dell'espressione. Lo standard non contempla espressioni simili alla 6.18, dichiarandole indefinite. Ciò non toglie che molti compilatori calcolano correttamente il valore di detta espressione: se $i = 9$ prima dell'espressione, dopo la sua esecuzione, il valore della variabile diventa $i = 11$.

[6]Dove per "oggetto" si intende una variabile o un'espressione.

Ben diverso è il caso della sottostante espressione:

$$i = (++i) + (++i) + 1; \tag{6.19}$$

In questo caso vengono violate in maniera plateale le linee guida della norma ISO/IEC 9899, ma in casi simili, contrariamente a ciò che può avvenire per la 6.18, nessun compilatore è in grado di garantire risultati prevedibili.

Lo studente deve prestare quindi estrema attenzione a quelle espressioni che modificano più d'una volta durante il calcolo, o più precisamente ad un determinato punto di sequenza, una determinata variabile/espressione. I risultati possono essere assolutamente impredicabili. Soprattutto possono essere forniti risultati diversi da compilatore a compilatore. Il che significa che il programma non è più *portabile*, ma dipende dall'ambiente di sviluppo.

6.5.2 Precedenza degli operatori di incremento/decremento

Parlare di precedenze a proposito degli operatori prefissi e postfissi rischia di essere pleonastico. Le due definizioni 9 e 10 hanno già illustrato in maniera chiarissima quale sia il livello di precedenza di detti operatori.

Il concetto verrà comunque ribadito con alcuni ulteriori esempi. Si suppogano i seguenti valori:

$$a = 1; \quad b = 2; \quad c = 3;$$

e le seguenti espressioni:

$$x = a + + * b + c; \tag{6.20}$$

$$y = + + a > b == a < b + +; \tag{6.21}$$

La prima espressione viene calcolata nel seguente modo:

$$x = 1 * 2 + 3;$$
$$x = 2 + 3;$$
$$x = 5;$$
$$a = 2;$$

Mentre la seconda ha il seguente sviluppo:

$$y = 2 > 2 == 2 < 2;$$
$$y = 0 == 0;$$
$$y = true;$$
$$a = 2;$$
$$b = 3;$$

6.6 Gli operatori bit a bit

Gli operatori *bit a bit* del linguaggio C sono "ereditati" direttamente dal linguaggio Assembly. Si tratta di operatori che operano direttamente su gruppi di bit *prescindendo dal significato che detto gruppo di bit racchiude*.

Ad esempio, una variabile di tipo intero potrebbe contenere il valore 100, ma operando bit a bit su tale variabile, si modificano *i singoli bit*, senza tener conto del significato (cioè il valore 100) che detti bit racchiudono tutti insieme.

Si possono classificare gli operatori bit a bit in tre fondamentali categorie:

- operatori logici bit-a-bit;
- operatore di complemento bit-a-bit;
- operatori di shift.

La prima categoria raggruppa i seguenti opearatori:

& | ^

che rappresentano rispettivamente l'operatore di AND a bit (*bitwise AND*), di OR a bit (*bitwise OR*) e di XOR a bit (*bitwise XOR*).

La seconda categoria comprende il solo operatore di complemento

~

che permette di eseguire il complemento a 1 di ciascun bit di una determinala variabile, ovvero rappresenta l'operatore di NOT a bit (*bitwise* NOT).

La terza categoria raggruppa i seguenti opearatori:

<< >>

che rappresentano rispettivamente l'operatore di SHIFT a sinistra (*bitwise left SHIFT*) e di SHIFT a destra (*bitwise right SHIFT*).

I singoli operatori meritano qualche piccola precisazione.

6.6.1 *Bitwise AND*

L'operatore AND a bit è un operatore binario ed opera sui singoli bit degli operandi. I due operandi devono essere entrambi interi.

La tabella di verità della funzione AND, per due bit, è indicata nella sottostante tabella:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Tabella 6.9: Funzione *bitwise* AND

Si supponga che venga eseguita la funzione AND a bit fra i seguenti valori:

01010101 A
11110000 B

01010000 A AND B

Se il primo operando vale 01010101 ed il secondo 11110000, la funzione AND a bit fra detti due operandi esegue la congiunzione logica dei singoli bit delle due variabili, ossia 01010000.

6.6.2 *Bitwise OR*

Anche l'operatore OR a bit è un operatore binario che opera sui singoli bit dei due operandi. Come per l'operatore AND, i due operandi devono essere entrambi interi.

La tabella di verità della funzione OR, per due bit, è indicata nella sottostante tabella:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Tabella 6.10: Funzione *bitwise* OR

Si supponga che venga eseguita la funzione OR a bit fra i seguenti valori:

```
01010101   A
11110000   B
________
11110101   A OR B
```

Se il primo operando vale 01010101 ed il secondo 11110000, la funzione OR a bit fra detti due operandi esegue la disgiunzione inclusiva dei singoli bit delle due variabili, ossia 11110101.

6.6.3 *Bitwise XOR*

L'operatore XOR a bit, come i precedenti, è un operatore binario che opera sui singoli bit dei due operandi. Entrambi gli operandi devono essere di tipo intero.

La tabella di verità della funzione XOR, per due bit, è indicata nella sottostante tabella:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 6.11: Funzione *bitwise* XOR

Si supponga che venga eseguita la funzione XOR a bit fra i seguenti valori:

```
01010101   A
11110000   B
________
10100101   A XOR B
```

Se il primo operando vale 01010101 ed il secondo 11110000, la funzione XOR a bit fra detti due operandi esegue la disgiunzione esclusiva dei singoli bit delle due variabili, ossia 10100101.

6.6.4 *Bitwise NOT*

E' l'unico operatore unario degli operatori bit a bit. Esso esegue il complemento a 1 della variabile su cui opera.

La tabella di verità della funzione NOT, per un bit, è indicata nella sottostante tabella:

A	NOT A
0	1
1	0

Tabella 6.12: Funzione *bitwise* NOT

Quindi il complemento a 1 del valore binario 01010101 diventa 10101010, ovvero la negazione booleana di ciascun bit.

6.6.5 *Bitwise left SHIFT*

Il termine *shift* significa *scorrimento*. Il *bitwise left SHIFT* esegue quindi uno scorrimento a sinistra dei bit di una determinata variabile. La fig. 6.1 illustra ciò che succede durante lo scorrimento a sinistra di *una posizione* dei bit di una determinata variabile, ad esempio, di 8 bit:

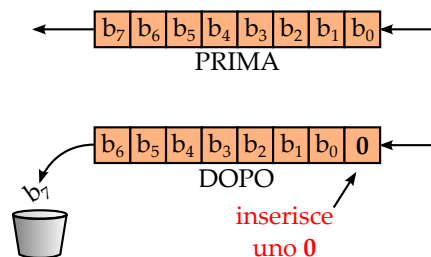



Figura 6.1: *Bitwise left SHIFT*

Si supponga, per semplicità, che prima dello scorrimento la variabile avesse valore binario 00000011, ossia 3_{10}. Dopo lo scorrimento a sinistra di una posizione il valore che la variabile assumerà sarà 00000110, ovvero 6_{10}, cioè il doppio.

Effettivamente, lo scorrimento a sinistra di n posizioni di una variabile intera di tipo *unsigned* coincide con la moltiplicazione della variabile per 2^n.Si è sottolineato l'attributo *unsigned* della variabile, perché man mano che si scorre la variabile a sinistra, se la variabile fosse *signed*, potrebbe essere alterato il segno della stessa, dato che essa è rappresentata in complemento a due. Si ponga attenzione, però che modificando il segno della variabile di tipo *signed* si *modifica totalmente ed irreversibilmente anche il valore assoluto della variabile*. Si vuole, quindi, sottolineare che il termine "moltiplicare" va interpretato *cum grano salis*.

Si noti che durante lo scorrimento, i bit più significativi vanno persi man mano che vengono ulteriormente *shiftati* oltre il bit più significativo della variabile. Contestualmente vengono introdotti degli zeri dai bit meno significativi.

Possibili usi dell'operatore di scorrimento a sinistra sono i seguenti:

Listing 6.6: Possibili usi dello SHIFT a destra

```
char a = 0x0F;
int b = 1;

a = a << b;     //Scorre a sinistra di b posizioni
a = a << 2;     // Scorre a sinistra di 2 posizioni
a = a << b*2+1; //Scorre del valore dell'espressione
```

6.6.6 *Bitwise right SHIFT*

Il *bitwise right SHIFT* esegue uno scorrimento a destra dei bit di una determinata variabile. La fig. 6.2 illustra ciò che succede durante lo scorrimento a destra di *una posizione* dei bit di una determinata variabile, ad esempio, di 8 bit:

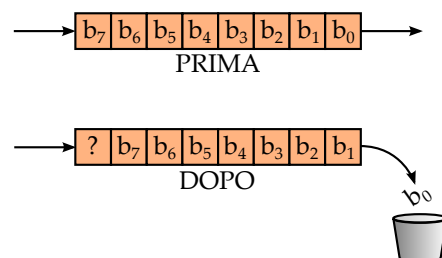



Figura 6.2: *Bitwise right SHIFT*

Si supponga, per semplicità, che prima dello scorrimento la variabile avesse valore binario 00000110, ossia 6_{10}. Dopo lo scorrimento a destra di una posizione il valore che la variabile assumerà sarà $X0000011$[7], ovvero 3_{10}, cioè la metà.

Effettivamente, lo scorrimento a destra di n posizioni di una variabile intera di tipo *unsigned* coincide con la divisione della variabile per 2^n.

[7]L'ambiguità del bit più significativo verrà affrontata tra breve. Momentaneamente si può ipotizzare che vengano inseriti degli zeri.

Si noti che durante lo scorrimento, i bit meno significativi vanno persi man mano che vengono ulteriormente *shiftati* oltre il bit meno significativo della variabile. Ciò che avviene contestualmente sui bit più significativi è leggermente più complesso che nello scorrimento a sinistra.

Ci si deve ricordare che le variabili intere possono essere rappresentate e trattate come *unsigned* oppure come *signed*, ovvero come variabili senza segno oppure con segno. In quest'ultimo caso esse sono rappresentate in complemento a due ed il bit più significativo rappresenta il segno del valore contenuto dalla variabile.

I due casi vanno mantenuti distinti, dato che lo scorrimento a destra (*non* lo scorrimento a sinistra, si noti: solo quello a destra!) opera sulla variabile in maniera diversa se la variabile è con segno oppure no.

Se la variabile è *signed* il risultato dello scorrimento a destra è illustrato dalla fig 6.3.

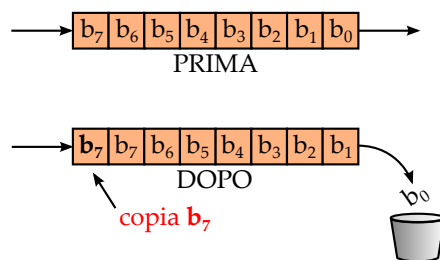



Figura 6.3: ***Signed*** *bitwise right SHIFT*

Nelle variabili *signed* vi è la necessità di *mantenere il segno*, il che obbliga a copiare il bit più significativo man mano che si esegue lo scorrimento a destra, indipendentemente dall'entità dello scorrimento.

Se la variabile è *unsigned* il risultato dello scorrimento a destra è illustrato dalla fig 6.4.

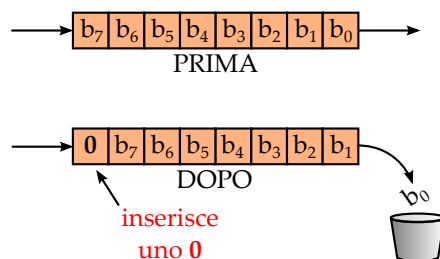



Figura 6.4: ***Unsigned*** *bitwise right SHIFT*

Nelle variabili *unsigned* non vi è alcuna necessità di mantenere il segno, il che permette di inserire degli zeri nel bit più significativo man mano che si esegue lo scorrimento a destra, indipendentemente dall'entità dello scorrimento.

Il programmatore deve porre estrema attenzione a questo apparentemente insignificante dettaglio: può essere frutto di innumerevoli dispiaceri[8]. A tal fine si deve ricordare che se la variabile intera è dichiarata senza l'attributo *unsigned*, automaticamente viene considerata come dotata di segno.

Quindi una variabile, ad esempio, definita di tipo `char` verrà considerata come dotata di segno durante lo scorrimento a destra.

I differenti effetti di uno scorrimento a destra su variabili con e senza segno sono illustrati nel sottostante esempio.

Listing 6.7: SHIFT a destra con e senza segno

```
char a = 0x8F;
unsigned b = 0x8F;

a = a >> 1;      //Si ottiene a = 0xC7
b = b >> 1;      //Si ottiene b = 0x47
```

Le forme sintattiche illustrate nel codice 6.6 a pagina 183 sono applicabili anche nello scorrimento a destra.

6.6.7 Associatività degli operatori bit a bit

Come già visto nei precedenti casi, gli operatori binari hanno associatività da sinistra verso destra, mentre quelli unari hanno associatività da destra verso sinistra. La tabella riassuntiva è quindi la seguente:

Operatore	Associatività
&	da sinistra verso destra
|	da sinistra verso destra
^	da sinistra verso destra
~	da destra verso sinistra
< <	da sinistra verso destra
> >	da sinistra verso destra

Tabella 6.13: Associatività degli operatori bit a bit

Si suppongano le seguenti variabili:

$$a = 1; \quad b = 2; \quad c = 3;$$

e la seguente espressione:

$$x = a << b|c; \tag{6.22}$$

L'espressione verrà svolta nel seguente modo:

$$\begin{aligned} &x = 1 << 2|3; && //1 << 2 \; equivale \; a \; 4 \; (00000100) \\ &x = 4|3; && //00000100 \; OR \; 00000011 \; = \; 00000111 \\ &x = 7; \end{aligned}$$

[8]Un caso eclatante è dato dal calcolo della parità polinomiale di un *frame* di dati. Un incauto uso dell'operatore di scorrimento può introdurre errori nel calcolo del CRC.

6.6.8 Precedenza degli operatori bit a bit

Anche per quanto riguarda gli operatori bit a bit sono rispettate le regole fondamentali già viste: gli operatori unari hanno la precedenza sugli operatori binari.

Definito ciò, gli operatori di scorrimento hanno la precedenza sui restanti operatori bit a bit.

Alla luce di quanto fin qui esposto, è quindi definibile la tabella delle precedenze 6.14. La collocazione dei suddetti operatori all'interno delle precedenze

Precedenza	Operatore
1	~
2	<< >>
3	& | ^

Tabella 6.14: Precedenza degli operatori bit a bit

globali è un po' più complessa e si preferisce fornire una tabella riassuntiva complessiva (vedi sez. 6.8).

6.7 Gli operatori di assegnazione

Nel cap. 3 è stato introdotto l'operatore di assegnazione "=". In realtà non si tratta dell'unico operatore che permette di assegnare dei valori a delle variabili. Si veda, ad esempio il seguente codice.

Listing 6.8: Operatori di assegnazione

```
int a = 0x55;
int b = 0xAA;

a = a + b;
a += b;          //Istruzione equivalente alla precedente
a = a * b;
a *= b;          //Istruzione equivalente alla precedente
a = a / b;
a /= b;          //Istruzione equivalente alla precedente
a = a % b;
a %= b;          //Istruzione equivalente alla precedente
a = a << b;
a <<= b;         //Istruzione equivalente alla precedente
a = a >> b;
a >>= b;         //Istruzione equivalente alla precedente
a = a & b;
a &= b;          //Istruzione equivalente alla precedente
a = a | b;
a |= b;          //Istruzione equivalente alla precedente
a = a ^ b;
a ^= b;          //Istruzione equivalente alla precedente
```

A due a due le istruzioni indicate nel codice sono assolutamente equivalenti. Le istruzioni commentate usano degli operatori di assegnazione che

permettono di rendere il codice più leggibile e più semplice. Detti operatori sono:

+= −= ∗= /= %= <<= >>= &= |= ^=

Detti operatori sono utilizzabili quando l'istruzione di assegnazione utilizzante semplicemente l'operatore "=" ha la seguente forma:

<esp1> = <esp1> op <esp2>

dove `<esp1>` deve essere una variabile e deve comparire immediatamente a destra e a sinistra dell'operatore "=".

6.8 Tabella riassuntiva

Alla luce di quanto detto nel presente capitolo si possono riassumere gli operatori nella sottostante tabella di associatività e precedenza. La precedenza decresce dall'alto verso il basso, mentre l'associatività è indicata nell'apposita colonna. Si noti che sono inclusi, per comodità di futura consultazione, anche degli operatori che non sono ancora stati affrontati.

Operatore	Associatività
( ) [ ] -> .	da sinistra verso destra
! ~ ++ -- + - *(type) &(add) sizeof	da destra verso sinistra
* / %	da sinistra verso destra
+ -	da sinistra verso destra
« »	da sinistra verso destra
< <= > >=	da sinistra verso destra
= = !=	da sinistra verso destra
&	da sinistra verso destra
^	da sinistra verso destra
|	da sinistra verso destra
&&	da sinistra verso destra
||	da sinistra verso destra
?:	da destra verso sinistra
= += -= *= /= %= «= »= &= |= ^	da destra verso sinistra
,	da sinistra verso destra

Tabella 6.15: Precedenza e associatività degli operatori

A proposito degli operatori elencati nella seconda riga della tabella, si rammenta che l'associatività degli operatori prefissi e postfissi cambia. Si veda a tal proposito la sez. 6.5.1.

6.9 Un po' di logica

Prima di affrontare il prossimo capitolo è bene rivedere sommariamente e senza molte pretese alcuni concetti di logica, più per fissare alcuni termini ricorrenti che per trattare l'argomento.

E' anche l'occasione per formalizzare alcuni concetti già introdotti sommariamente ma ai quali non si è dato dignità formale, nonché per acquisire alcuni strumenti utili per affrontare il prossimo capitolo in maniera consapevole.

La logica è quella branca della matematica che studia il modo di trarre delle conclusioni, attraverso un particolare tipo di ragionamento, detto deduttivo. Secondo Piergiorgio Odifreddi , essa è semplicemente la scienza del ragionamento (cfr. ODIFREDDI [20]). Tale definizione, particolarmente efficace, racchiude due pilastri della logica matematica, indicando sia di "cosa" la logica si occupi (ossia del ragionamento) e di "come" se ne occupi (attraverso il metodo scientifico e più specificatamente attraverso il metodo assiomatico tipico della matematica).

Un uso corretto della logica matematica ci permetterà di porre ordine nei nostri ragionamenti, limitando gli errori e le ridondanze o addirittura i paradossi, come dimostra il celebre sillogismo di Montaigne: "Il salame fa bere, bere è dissetante, dunque il salame disseta". Soprattutto, però, ci permetterà di non dipendere dalla fantasia e dall'ispirazione di un momento, ma di applicare un metodo rigoroso e scientifico che sarà d'aiuto sempre, anche nei momenti di scarsa ispirazione.

Infine, per chi si avvicina per la prima volta allo studio dei linguaggi di programmazione, la logica (e specificatamente la logica matematica) offre ulteriori spunti di riflessione perché, oltre a fornire delle conoscenze che si sono già definite imprescindibili, permette di condividere un linguaggio comune e non ambiguo. Quest'ultimo aspetto che è frequentemente sottovalutato dallo studente è, invece, un elemento di fondamentale importanza.

6.9.1 Le proposizioni

In logica matematica si definisce **proposizione** un enunciato, espresso in linguaggio naturale, formale o simbolico, del quale si possa dire, senza ambiguità alcuna e con assoluta certezza, se esso sia **Vero** o **Falso**, dove Vero e Falso sono detti valori di verità.

Sono proposizioni le seguenti frasi:

- Roma è la capitale d'Italia
- $1 + 1 = 3$
- Tu se' lo mio maestro
- Piove

La prima proposizione è vera, mentre la seconda proposizione è falsa (ma resta pur sempre una proposizione). La terza proposizione è vera, ma potrebbe essere anche falsa. L'importante è che sia vera o falsa senza ambiguità. Noi la accettiamo per vera (ci mancherebbe: lo dice Dante). Per sapere se la quarta proposizione è vera o falsa bisogna guardare fuori dalla finestra. Ma non è questo il punto: a noi basta sapere che o piove (valore di verità Vero) o non piove (valore di verità Falso).

Le seguenti, invece, non sono proposizioni:

- Pierino è abbastanza bello
- $2+3$ [9]
- Forza Juve (o Udinese, o Milan, o Inter, o Albinoleffe, ecc.)
- Passami la clava

La prima affermazione è vaga e ambigua, a causa della presenza dell'avverbio "abbastanza". La seconda frase è un'espressione aritmetica, mentre in termini matematici interessano le espressioni logiche. La terza frase può creare accesi dibattiti e dispute accalorate, ma non può essere vera o falsa. Anche l'ultima frase presenta le stesse caratteristiche, anche se prudenzialmente, è sempre meglio non passare la clava a nessuno.

Il fatto che le proposizioni possano essere solo Vere o False porta direttamente alla formulazione di due princìpi fondamentali della logica matematica:

Principio 1 (Principio di Non Contraddizione)**.**
Una proposizione non può essere contemporaneamente vera e falsa: l'una esclude l'altra.

Principio 2 (Principio del Terzo Escluso)**.**
Una proposizione può essere solamente o vera o falsa: non esiste una terza possibilità.

Le proposizioni possono essere **semplici** (atomiche) o **composte**. Sono semplici se sono formate da un solo enunciato e sono composte se sono formate da più enunciati, legati fra loro mediante dei **connettivi logici**. Nel linguaggio C tali connettivi sono chiamati operatori logici (vedi sez. 6.4).

Le proposizioni presentate nella pagina precedente sono tutte proposizioni semplici. Sono, invece, proposizioni composte le seguenti:

- Pierino mangia o beve
- $1+1=3 \vee 1+1=2$
- Tu se' lo mio maestro e 'l mio autore
- Piove e grandina

ma anche

- Se piove, prendo l'ombrello
- Prendo l'ombrello se e solo se piove

che non sono equivalenti dal punto di vista della logica matematica, come si vedrà nelle prossime pagine. In particolare, i connettivi logici che reggono le ultime due proposizioni composte verranno analizzati tra breve, perché non ancora trattati esplicitamente nel capitolo precedente.

[9] Si è già accennato che, in realtà, il linguaggio C riconosce tale espressione come proposizione, indicandola come *non falsa* per non entrare in contraddizione con il principio del terzo escluso, per cui in contesto matematico si può interpretare come abbreviazione sintattica della proposizione $2+3 \neq 0$.

6.9.2 I predicati

Un **predicato** è una relazione [10] in n variabili. Un esempio di predicato in una variabile è *x mangia la mela*. La particolarità del predicato è che non si sa a priori se esso sia vero o falso, perché non si conosce il valore della variabile.

Nel momento in cui la variabile assume un valore appartenente al proprio dominio (supponendo, ad esempio, che $x = Pierino$), allora il predicato diventa proposizione e di esso si può stabilire il valore di verità. Nel caso dell'esempio si ha: *Pierino mangia la mela*, di cui è possibile stabilire se è vera o falsa.

Un esempio di predicato in due variabili (in tal caso si ha una **relazione binaria**) è: *x è la capitale di y*. Anche in questo caso è impossibile stabilire il valore di verità del predicato, perché non si conosce il valore di x e di y. Nel momento in cui si stabilisce, ad esempio, che $x = Roma$ e $y = Germania$ si ottiene una proposizione falsa.

6.9.3 I connettivi logici

Nel paragrafo 6.9.1 si è accennato ai connettivi logici. Essi sono degli operatori che cambiano il valore di verità delle proposizioni o le legano fra loro. Come già detto a proposito degli operatori logici, essi si distinguono in **unari** e **binari** a seconda che abbiano rispettivamente uno o due argomenti.

Essi sono:

- la **negazione**. Operatore unario. Simbolo matematico: $\neg$;
- la **congiunzione**. Operatore binario. Simbolo matematico: $\wedge$;
- la **disgiunzione inclusiva**. Operatore binario. Simbolo matematico: $\vee$;
- la **disgiunzione esclusiva**. Operatore binario. Simbolo matematico: $\dot{\vee}$;
- la **implicazione**. Operatore binario. Simbolo matematico: $\rightarrow$;
- la **doppia implicazione**. Operatore binario. Simbolo matematico: $\leftrightarrow$.

Si sono già stabilite le correlazioni fra i suddetti connettivi logici e i relativi operatori del linguaggio C nelle sezioni precedenti, fuorché per l'implicazione e la doppia implicazione. Argomento che verrà affrontato fra poco.

6.9.3.1 L'implicazione

L'implicazione (o implicazione materiale) è un operatore binario. Date due proposizioni p e q, la nuova proposizione composta $p \rightarrow q$ è falsa solo se p è vera e q è falsa. Negli altri casi essa è sempre vera. Nel linguaggio naturale l'implicazione materiale è identificata con la dicitura "*se .. allora*" e l'espressione logica $p \rightarrow q$ si legge "*se p allora q*". La proposizione p è detta **precedente** e la proposizione q è detta **conseguente**.

[10] Si ricorda che la relazione n-aria è un qualsiasi sottoinsieme del prodotto cartesiano delle n variabili

Si fa notare che mentre nel linguaggio naturale una frase del tipo "se p allora q" implica una consequenzialità fra la causa p e l'effetto q, nel linguaggio matematico ciò *non è assolutamente vero*.

Si potrebbe, infatti, scrivere dal punto di vista matematico: "Se Parigi è la capitale della Francia allora il Po è il fiume più lungo d'Italia". Siccome entrambe le proposizioni sono vere, la proposizione composta è anch'essa vera. Essa, però, è totalmente priva di significato nel linguaggio naturale.

L'implicazione materiale non ha quindi correlazione logica con la lingua parlata. Questo particolare, molto importante, è da tener presente quando si tratterà l'argomento dal punto di vista del linguaggio C. Vedremo infatti che l'implicazione materiale come è solitamente espressa in linguaggio naturale (*se .. allora*) assomiglia moltissimo alla *struttura di selezione if .. else* che verrà trattata nel prossimo capitolo. Sarà importante non creare confusioni fra l'implicazione materiale e la struttura di selezione.

Esempi di implicazione materiale sono i seguenti:

p	q	$p \to q$
Piove	Prendo l'ombrello	**Se** piove **allora** prendo l'ombrello
$1 + 1 = 3$	$2 + 2 = 4$	**Se** $1 + 1 = 3$ **allora** $2 + 2 = 4$

Tabella 6.16: Esempi di implicazione materiale

Si nota che la prima proposizione ha attinenza logica anche con la lingua parlata, mentre la seconda sembra essere priva di senso. In realtà si tratta di una normale implicazione vera, dato che la proposizione precedente è falsa e la conseguente è vera (vedi tabella di verità posta di seguito).

La tabella di verità della implicazione assume la seguente forma:

p	q	$p \to q$
Falso	Falso	Vero
Falso	Vero	Vero
Vero	Falso	Falso
Vero	Vero	Vero

Tabella 6.17: Tabella di verità della implicazione

Si ribadisce ancora che l'operatore implicazione non esiste nel linguaggio C e non va confuso con la struttura di selezione che verrà trattata nel prossimo capitolo. Tale confusione nasce dal fatto che il costrutto sintattico del linguaggio naturale con il quale si esprime sia l'implicazione materiale che la selezione sia in entrambi i casi la forma *se .. allora*.

6.9.3.2 La doppia implicazione

La doppia implicazione (o equivalenza logica) è un operatore binario. Date due proposizioni p e q, la nuova proposizione composta $p \leftrightarrow q$ è vera solamente se $p = q$. Negli altri casi essa è falsa.

Nel linguaggio naturale l'implicazione materiale è identificata con la dicitura "*se e solo se .. allora*" e l'espressione logica $p \leftrightarrow q$ si legge "*se e solo se p allora q*".

Anche in questo caso la mente cerca di stabilire una connessione con quanto appreso studiando il linguaggio C. In questo caso, visto che la doppia implicazione è una *equivalenza logica* è più facile correlare la Logica con l'Informatica, come si vedrà tra breve.

Gli stessi esempi formulati nel paragrafo precedente assumono ora valori diversi. Si veda a tal proposito la tabella 6.18. Infatti la seconda proposizione, che risultava vera se valutata come implicazione materiale, ora diventa falsa se valutata come doppia implicazione. Ciò diventa evidente se si valutano le singole proposizioni semplici con la tabella di verità della doppia implicazione.

p	q	$p \leftrightarrow q$
Piove	Prendo l'ombrello	**Se e solo se** piove **allora** prendo l'ombrello
1 + 1 = 3	2 + 2 = 4	**Se e solo se** 1 + 1 = 3 **allora** 2 + 2 = 4

Tabella 6.18: Esempi di doppia implicazione

La tabella di verità della doppia implicazione assume la seguente forma:

p	q	$p \leftrightarrow q$
Falso	Falso	Vero
Falso	Vero	Falso
Vero	Falso	Falso
Vero	Vero	Vero

Tabella 6.19: Tabella di verità della doppia implicazione

Ovviamente essendo la doppia implicazione indicata anche come operatore di equivalenza, si può stabilire una correlazione l'analogo operatore del linguaggio C visto nella sezione 6.3.

Prima di affrontare il prossimo importante capitolo, conviene, però, ricordare due fondamentali principi matematici, che torneranno molto utili quando si tratterà di formulare espressioni logiche corrette ed efficienti.

6.9.4 Le leggi di De Morgan

Il matematico inglese Augustus De Morgan (1806 - 1871), contemporaneo di George Boole, formulò due importanti leggi:

Legge 1 (Prima legge di De Morgan)**.**
La negazione di una congiunzione logica è uguale alla disgiunzione inclusiva fra le proposizioni negate. In simboli:

$$\neg(p \wedge q) = \neg p \vee \neg q \qquad \overline{p \wedge q} = \overline{p} \vee \overline{q} \tag{6.23}$$

Legge 2 (Seconda legge di De Morgan)**.**
La negazione di una disgiunzione inclusiva è uguale alla congiunzione fra le proposizioni negate. In simboli:

$$\neg(p \vee q) = \neg p \wedge \neg q \qquad \overline{p \vee q} = \overline{p} \wedge \overline{q} \tag{6.24}$$

I due suddetti principi sono la diretta evoluzione del lavoro svolto da George Boole più o meno negli stessi anni e costituiscono due pilastri della logica matematica.

Le due leggi si provano facilmente confrontando le due tabelle di verità:

p	q	$p \wedge q$	$\overline{p \wedge q}$
Falso	Falso	Falso	Vero
Falso	Vero	Falso	Vero
Vero	Falso	Falso	Vero
Vero	Vero	Vero	Falso

Tabella 6.20: Tabella di verità della negazione di una congiunzione logica

L'ultima colonna a destra fornisce i valori di verità della negazione di una congiunzione logica. E' facile verificare che l'ultima colonna a destra della tabella di verità della disgiunzione inclusiva delle proposizioni negate è assolutamente identica:

p	q	$\overline{p}$	$\overline{q}$	$\overline{p} \vee \overline{q}$
Falso	Falso	Vero	Vero	Vero
Falso	Vero	Vero	Falso	Vero
Vero	Falso	Falso	Vero	Vero
Vero	Vero	Falso	Falso	Falso

Tabella 6.21: Tabella di verità della disgiunzione inclusiva delle proposizioni negate

In tal modo si è dimostrata la prima legge di De Morgan. In maniera del tutto analoga si dimostra la seconda legge.

Le leggi di De Morgan, insieme alle leggi di assorbimento e alle proprietà distributive della congiunzione rispetto alla disgiunzione inclusiva e della disgiunzione inclusiva rispetto alla congiunzione (che verranno esaminate

nel prossimo paragrafo), assumono grande importanza sia in elettronica digitale che nei linguaggi di programmazione. Lo studente vi ponga, quindi, particolare attenzione.

Un'applicazione pratica (potenzialmente utile agli adolescenti particolarmente vivaci) delle leggi di De Morgan, anche se obiettivamente posta a metà strada fra lo scherzo ed il paradosso, è data dal seguente

Esercizio - ◊◊♦ Smettila di saltare e urlare

Il padre di Pierino, stufo del proprio figlio che "salta **e** urla", per ottenere un po' di calma potrebbe intimargli un perentorio "Smettila di saltare **e** urlare". Potrebbe, però, trattarsi di un errore clamoroso nel caso in cui Pierino sia stato attento durante le lezioni di Matematica e di Informatica, quando i suoi insegnanti hanno spiegato le leggi di De Morgan.

Soluzione

Il figlio potrebbe, infatti, accogliere i desideri paterni traducendoli logicamente in "Pierino non (deve) saltare e urlare", giocando abilmente e in maniera ambigua sul ruolo della negazione. Assegnando a p il valore "Pierino salta" ed a q il valore "Pierino urla", si ottiene in simboli:

$$\neg(p \wedge q) \tag{6.25}$$

Applicando la prima legge di De Morgan si ottiene:

$$\neg p \vee \neg q \tag{6.26}$$

che equivale alla proposizione "Pierino non salta o non urla", ovvero, rendendola sintatticamente più appropriata "Pierino non (deve) saltare o non (deve) urlare". Il figlio potrebbe quindi smettere di saltare continuando ad urlare come un ossesso pur avendo obbedito al genitore (avente lacunose conoscenze di logica matematica).

Il genitore, probabilmente, vorrebbe che il proprio figlio smettesse di saltare e smettesse di urlare, ovvero, in simboli:

$$\neg p \wedge \neg q \tag{6.27}$$

quindi farebbe bene a dire al proprio figlio "Smettila di saltare **e** smettila di urlare", oppure potrebbe esortarlo, in maniera piuttosto goffa, ma con l'approvazione di Augustus De Morgan, nel seguente modo: "Smettila di, parentesi aperta, saltare **o** urlare, parentesi chiusa" [11].

6.9.5 Un altro esempio

Lo studente ha sicuramente colto, durante lo svolgimento del precedente esercizio, la difficoltà di una corretta "traduzione" del testo fornito in linguaggio naturale in una serie di proposizioni elementari appropriatamente collegate

[11] Si consiglia agli studenti, comunque, di cogliere, in via del tutto eccezionale, il senso delle genitoriali richieste anche nel caso siano formulate in maniera poco ortodossa dal punto di vista della logica matematica.

fra loro mediante i corretti connettivi logici. Tale difficoltà è intrinsecamente legata al linguaggio naturale e solamente un'attenta analisi e una certa confidenza con il linguaggio formale d'arrivo possono soccorrere lo studente in tale compito.

Si tratta di un'azione tutt'altro che facile alla quale George Boole ha dedicato un'importante parte della propria vita e del proprio impegno intellettuale. Non si pretende, quindi, di fornire strumenti adeguati allo studente attraverso un banale esercizio. Si vuole, piuttosto, far notare come un atteggiamento rigoroso e metodico nei confronti di un problema dato produca sempre dei frutti. Con il tempo (e con la perseveranza dello studente) tali frutti matureranno.

Per tutti questi motivi, si ritiene quindi utile proporre un secondo esercizio sul tema. Si tratterà, in particolare dell'interpretazione e del confronto fra due proposizioni.

Esercizio - ◊◊♦ Confronto fra proposizioni

Il seguente esercizio vuole porre in evidenza la difficoltà della "traduzione" e confronto di due proposizioni formulate in linguaggio naturale.

Ma quanto mi costi?

"I prodotti di qualità sono costosi" ed "I prodotti costosi sono di qualità". Si valuti se le due proposizioni sono logicamente equivalenti.

Si assuma che "costoso" significhi "avente costo maggiore di..." e che "di qualità" significhi "conforme alla normativa...".

Soluzione

Si è riusciti a rendere le due proposizioni non ambigue fornendo le due precisazioni nella seconda parte del testo. Si tratta ora di tradurle nel linguaggio della logica matematica e di confrontarle. A tal fine è bene cercare di far emergere quale sia il reale significato di cui le due frasi sono portatrici.

Una delle possibili interpretazioni è la seguente:

Versione 1.01

"**Se** i prodotti sono di qualità **allora** sono costosi" e "**Se** i prodotti sono costosi **allora** sono di qualità".

Ora è possibile distinguere in entrambi i casi una implicazione materiale formate da una proposizione precedente ed una conseguente. Attribuendo p a "i prodotti sono costosi" e q a "i prodotti sono di qualità", è possibile formulare la seguente espressione:

$$p \to q = q \to p \tag{6.28}$$

e confrontando la suddetta espressione con la tabella di verità dell'implicazione materiale (vedi tab. 6.17 a pagina 191) dedurre che quando le proposizioni p e q hanno gli stessi valori di verità, l'uguaglianza risulta essere verificata.

Se, però, le due proposizioni assumono valori di verità diversi , ad esempio $p = Vero$ e $q = Falso$, si ottiene che l'uguaglianza non è verificata:

$$Vero \rightarrow Falso = Falso \rightarrow Vero \tag{6.29}$$
$$Falso = Vero \tag{6.30}$$

Ciò significa che le due proposizioni composte originali non sono equivalenti. Molte massaie sono giunte alle stesse conclusioni senza l'aiuto della logica matematica.

E' bene però porre in guardia lo studente. Sicuramente lo studente ha colto lo spirito dell'esercizio e ha volentieri "giocato" cercando di risolverlo. Dal punto di vista strettamente matematico, però, c'è stata una evidente forzatura: si è "interpretato" prima il senso letterale del testo (aggiungendo il *se .. allora*), ignorando la sua matrice matematica, e poi risolvendolo dal punto di vista logico come implicazione materiale. In tal modo, però, si è deformato proprio il significato del *se .. allora*.

Insomma, l'autore deve ammettere di avere barato un po', stavolta. D'altronde, il presente voleva essere solamente un esercizio. Lo studente rimanga, però, sempre critico davanti ad una implicazione materiale e ricordi che per essa non vale la "interpretazione reale" del linguaggio naturale.

Se si volesse tentare di dare al problema originale una maggiore attinenza con il linguaggio naturale si potrebbe tentare di riformularlo nel seguente modo:

Versione 1.02

"**Se e solo se** i prodotti sono di qualità **allora** sono costosi" e "**Se e solo se** i prodotti sono costosi **allora** sono di qualità".

Si tratta di una formulazione piuttosto pedante dal punto di vista del linguaggio naturale ma semplicemente corretto dal punto di vista logico-formale.

Se le suddette proposizioni vengono prese alla lettera, dal punto di vista del linguaggio naturale si lascia spazio a discussione, non vi è alcun dubbio. La discussione, però, dovrebbe basarsi su opinioni personali e solo marginalmente su considerazioni oggettive.

Comunque, se riformulate come nella versione 1.02, le due proposizioni sono assolutamente equivalenti data la simmetricità della tabella di verità della doppia implicazione. Lo studente può facilmente verificare quanto appena sostenuto.

6.10 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 6.

Capitolo 7

Le istruzioni condizionali

"Naturally, **if** *I had been unable to sleep,*
if *I had mounted the stairs,*
then perhaps I might have seen this assassin,
this monster enter or leave Madame's cabin,
but as it is."

Death on the Nile (1937) - Agatha Christie

Le strutture decisionali ricoprono un ruolo fondamentale nella programmazione strutturata. Per capire meglio il loro ruolo si devono offrire alcuni scampoli di storia dell'Informatica.

La naturale esecuzione di un programma è quella *sequenziale*, ove le singole istruzioni sono eseguite una dopo l'altra in sequenza. A volte è però necessario analizzare determinati dati e *prendere delle decisioni*, ossia eseguire determinate istruzioni piuttosto che altre, interrompendo in tal modo l'esecuzione sequenziale. Tale necessità è detta *trasferimento di controllo* e negli anni '60 veniva operata mediante un test condizionale e successiva esecuzione dell'istruzione `goto`.

L'uso spesso sconsiderato di detta istruzione accese, in quegli anni, una vivace discussione nell'ambiente informatico, che mise sotto accusa il `goto` e fece nascere il concetto di *programmazione strutturata*. Quest'ultimo coincise, praticamente, con l'eliminazione della perfida istruzione (cfr. DEITEL & DEITEL[2]) e talvolta anche con la sua ridicolizzazione, come appare evidente dal pensiero di Ed Post che campeggia nell'Introduzione:

"Real Programmers aren't afraid to use GOTOs."

Nella seconda metà degli anni '60 due ricercatori italiani, Corrado Böhm e Giuseppe Jacopini, formularono un famoso teorema, detto *Teorema di separazione* o *Teorema di Böhm-Jacopini*, mediante il quale venne dimostrato che era possibile scrivere un qualsiasi programma senza l'uso dei `goto`.

Dimostrarono che un qualsiasi programma poteva essere formato mediante sole tre strutture di controllo: la *sequenza*, la *selezione* e l'*iterazione*. La sequenza è già stata parzialmente illustrata nei precedenti capitoli, mentre la selezione (o istruzione condizionale oppure struttura decisionale) è l'oggetto del capitolo presente. Quello prossimo tratterà delle iterazioni.

Graficamente la struttura decisionale è la seguente:

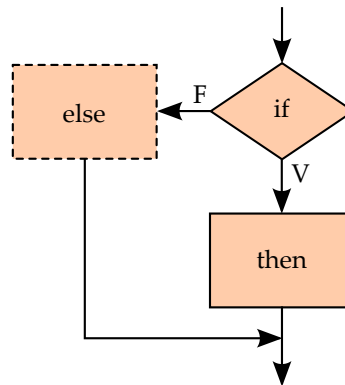



Figura 7.1: Struttura decisionale

Il rombo rappresenta la *condizione booleana*, che può essere *vera* oppure *falsa*. Una qualsiasi struttura decisionale inizia, quindi, sempre con un *test*, dal cui esito dipende il prosieguo del programma: se la condizione booleana risulta essere vera, verrà eseguito il *corpo dell'if*, ovvero il blocco al cui interno è stato scritto "then"[1], altrimenti verrà eseguito, se esiste, il *corpo dell'else*, ovvero il blocco al cui interno è stato scritto "else".

In linguaggio C l'istruzione condizionale assume la seguente forma:

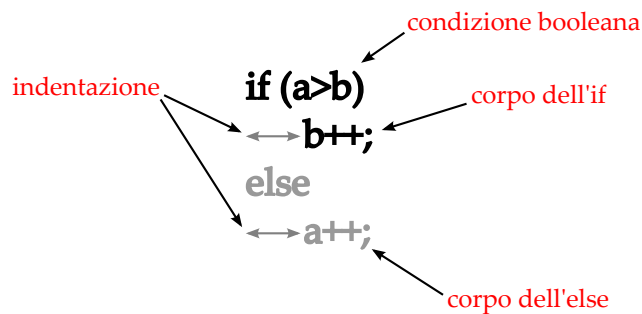



Figura 7.2: Istruzione if (else)

La figura si presta a numerose considerazioni. Analizziamole quindi una ad una.

La prima riga contiene la la condizione booleana. La sintassi è la seguente

`if (`<condizione booleana>`)`

[1]Si noti che "then" non è una parola chiave. Si tratta semplicemente di un termine mutuato dal Pascal ed è stato usato perché si ritiene aggiunga chiarezza al costrutto.

Il primo elemento dell'istruzione è la parola chiave `if` seguita dalla condizione booleana racchiusa fra parentesi tonde. Le parentesi *non sono opzionali*. La condizione booleana deve osservare le regole già indicate a proposito degli operatori aritmetico-logici e deve restituire un valore che sarà interpretato come valore di verità.

A tal proposito si deve osservare che, al fine d'osservare il principio del terzo escluso, si può fare solamente *un* test per verificare il valore di verità: l'istruzione `if` verifica solamente se la condizione booleana è *non falsa*, ovvero se il valore ritornato dalla condizione booleana è *diverso da zero*. In tal caso l'istruzione `if` assumerà che, non essendo zero, la condizione booleana sarà considerata (indirettamente) *vera*. E' importante notare che non viene effettuato alcun test per verificare direttamente se la condizione booleana è vera.

Lo studente deve aver ben chiaro questo dettaglio: le implicazioni elencate nel capitolo precedente (vedi sez.6.4) sono di estrema importanza e possono creare numerosi grattacapi al programmatore.

7.1 L'istruzione *if*

Con riferimento alla fig. 7.2, la seconda riga rappresenta il *corpo dell'if*. Questa parte di programma viene eseguita solamente se la condizione booleana è *non falsa*. A seconda delle esigenze del programma, il corpo dell'if può essere formato da una istruzione oppure da più istruzioni. In questo secondo caso il corpo dell'if *deve essere racchiuso fra parentesi graffe*, come nel sottostante esempio:

Listing 7.1: Esempio di corpo dell'if

```
if (a>b)
{               //Si notino le parentesi graffe...
   a++;         //...per racchiudere il...
   b += a+2;    //...corpo dell'if
}
```

Il costrutto relativo al codice 7.1 è denominato *istruzione if* oppure *struttura di selezione if* oppure genericamente *istruzione condizionale*.

Si noti che il corpo dell'if è *indentato*, il che significa che le istruzioni che formano il corpo dell'if sono rientrate di qualche spazio (solitamente 3-4 spazi) rispetto all'espressione condizionale. Si tratta di una buona norma che lo studente farebbe bene ad osservare da subito. In tal modo il programma risulta essere molto più leggibile, come pure diventa più facile trovare eventuali errori.

Nel caso in cui la condizione booleana dovesse risultare *falsa* il corpo dell'if semplicemente non verrebbe eseguito.

7.2 L'istruzione *if..else*

La struttura di selezione possiede anche un secondo costrutto, denominato *istruzione if..else* oppure *struttura di selezione if..else* oppure nuovamente *istruzione condizionale*. Esso assume la seguente forma:

Listing 7.2: Struttura di selezione if..else

```
if (a)
{
   a++;         //Corpo dell'else...
   b += a+2;    //...racchiuso fra graffe
}
else
{               //Si notino le parentesi graffe...
   a--;         //...per racchiudere il...
   b -= a-2;    //...corpo dell'else
}
```

Il corpo dell'else è indentato esattamente come quello dell'if, a rimarcare la stessa struttura gerarchica. Come per quanto avvenuto per il corpo dell'if, il corpo dell'else è stato racchiuso fra parentesi graffe.

Le azioni che vengono eseguite dall'istruzione `if..else` sono leggermente diverse da quelle eseguite dall'istruzione `if`: la `if..else` esegue il corpo dell'if se la condizione booleana *a* è *non falsa*, mentre esegue il corpo dell'else se la condizione booleana è *falsa*. Ciò è equivalente a dire, nell'esempio specifico, che l'istruzione esegue il corpo dell'if se *a* è diverso da zero ed esegue il corpo dell'else se *a* è uguale a zero.

Naturalmente è possibile che il corpo dell'else sia formato da un'unica istruzione. In tal caso le parentesi graffe sono superflue (ma non errate).

Frequentemente lo studente indica il costrutto di fig.7.2 come *ciclo if*. Si tratta di un orribile ossimoro! Il codice di fig.7.2 è un'*istruzione condizionale*. Alternativamente essa può essere indicata mediante uno degli altri modi già illustrati, ma assolutamente ***mai come ciclo if***.

7.3 Le strutture *if..else* nidificate

Naturalmente, sia il corpo dell'if che il corpo dell'else possono contenere ulteriori istruzioni condizionali, dando in tal modo vita a strutture `if..else` *nidificate*. Un esempio di nidificazione, molto elegantemente redatta ma di non immediata leggibilità, è illustrato nel sottostante codice:

Listing 7.3: Strutture if..else nidificate

```
if (a>b)        //Il codice e' volutamente...
   b++;         //...non commentato dal...
else if (a>c)   //...punto di vista semantico.
   c++;         //Qual e' la condizione...
else if (a>d)   //...booleana che, se falsa,...
   d++;         //...conduce all'incremento...
else            //...della variabile a?
   a++;
```

La scrittura del codice 7.3, talvolta anche indicato come struttura *else..if*, è effettivamente quella preferita dai programmatori: stilisticamente molto elegante, senza "tendere" eccessivamente troppo a destra a causa delle continue indentazioni. Ad un primo sguardo può presentare, però, cenni di ambiguità, soprattutto all'occhio di uno studente alle prime armi.

Ci si potrebbe chiedere legittimamente quale sia la condizione booleana che permette l'esecuzione dell'istruzione posta in ultima riga, ossia il postincremento della variabile *a*.

Se la condizione booleana `a>b` è falsa, viene eseguito il secondo `if`, cioè viene testata l'espressione `a>c`. Se anche quest'ultima è falsa, viene eseguito il terzo `if` e testata l'espressione `a>d`. Se anche questa risulta essere falsa viene finalmente eseguito l'incremento di *a*. La condizione booleana che permette l'esecuzione del postincremento di *a* è quindi:

$$!(a > b)\&\&!(a > c)\&\&!(a > d) == true \tag{7.1}$$

Il codice 7.3 può essere riscritto in forma più chiara nel seguente modo, assolutamente equivalente al precedente:

Listing 7.4: Forma equivalente

```
if (a>b)
   b++;
else
{               //Inizio 1^else
   if (a>c)
      c++;
   else
   {            //      Inizio 2^else
      if (a>d)
         d++;
      else
         a++;   //              3^else
   }            //      Fine 2^else
}               //Fine 1^else
```

L'unico difetto di tale codice consiste nel tendere a destra progressivamente, man mano che si aggiungono livelli di nidificazione.

7.4 Un esempio d'uso delle strutture di selezione

E' venuto il momento di affrontare un esercizio che utilizzi tutto quanto fin qui imparato: l'*input/output* da tastiera e verso il video; l'uso degli operatori aritmetico-logici; l'uso delle assegnazioni; l'interpretazione e "traduzione" del testo; l'uso delle strutture di selezione.

A tal fine si propone il seguente

Esercizio - ◊◊♦ Metodo di valutazione anno bisestile

Si fornisce di seguito, in linguaggio naturale, un possibile metodo per valutare se un determinato anno è bisestile o meno:

Metodo di valutazione anno bisestile

Un anno è bisestile se è divisibile per 4, ma non per 100, a meno che non sia divisibile per 400, ma non per 4000.

Si chiede di

1. comprendere il testo;
2. tradurlo in espressione logica matematica;
3. fornire un algoritmo di valutazione dell'anno bisestile indipendente dall'espressione matematica illustrandolo mediante un diagramma di flusso;
4. scrivere il codice in linguaggio C che formalizzi il diagramma di flusso di cui al punto 3.

Soluzione

Analizzare il testo in questione implica un po' di pazienza e una montagna di domande da porsi. E' lecito formulare le domande in qualsiasi modo, ma esse vanno logicamente interpretate e la risposta deve essere coerente con i principi, le leggi, le proprietà, ecc. della logica matematica.

Innanzi tutto, ci si deve chiedere se il testo è da considerarsi una proposizione o se ci sono parti che lo invalidano in quanto tale. Se così fosse sarebbe opportuno apportare delle modifiche al testo, senza modificarne il significato, in modo da renderlo ortodosso dal punto di vista logico.

Colpisce, ad esempio, l'iniziale "Un" che rende alquanto vaga la prima parte del testo. La frase "Un anno è bisestile" non può dirsi, infatti, una proposizione, dato che l'estensore del testo, probabilmente intendeva: "L'anno tal dei tali è bisestile", dove "tal dei tali" può essere interpretato come 2012, 1999, ecc.

L'espressione "tal dei tali" indica quindi una variabile e la frase in cui essa è inserita è un predicato. Si devono quindi operare alcune modifiche al testo originale, in modo da renderlo più ortodosso e, possibilmente, privo di ambuiguità, al fine di poter operare quella "traduzione" che ci permetterà di analizzare il testo dal punto di vista logico.

Una seconda versione del testo, più logica e meno discorsiva potrebbe essere la seguente:

Versione 1.01

L'anno x è bisestile se è divisibile per 4, ma non per 100, a meno che non sia divisibile per 400, ma non per 4000.

Ora le proposizioni atomiche *sembrano* tutte corrette, il che significa che la proposizione composta (cioè l'intero testo) può essere solo Vera o Falsa (principio del Terzo Escluso).

Analizzando un po' più approfonditamente il testo, però, si rileva la presenza di un "se" e vien da chiedersi, quindi, se si è in presenza di una implicazione materiale o meno.

In realtà, chi ha formulato il testo intendeva sottolineare una sostanziale equivalenza fra la proposizione "L'anno x è bisestile" ed il resto della frase, che indica, proprio, il metodo per valutare se è bisestile o meno.

Se, determinato un anno, è vera la proposizione precedente, deve essere vera anche la proposizione conseguente e, analogamente, se è falsa la precedente deve essere falsa anche la conseguente.

Questo ci porta, però, a concludere che si è in presenza di una doppia implicazione e non di una implicazione materiale e che, nuovamente, il testo non è ortodosso dal punto di vista logico.

Si propone, quindi un'ulteriore versione:

Versione 1.02

L'anno x è bisestile se e solo se x è divisibile per 4, ma non per 100, a meno che non sia divisibile per 400, ma non per 4000.

Il testo appare decisamente migliorato anche se ci sono ancora dei "ma" e degli "a meno che", che rendono ancora, a tratti, difficoltosa la traduzione. Per completare il processo, conviene, quindi, addentrarsi maggiormente nella comprensione semantica del testo.

Innanzi tutto si nota subito che, dati i numeri 4, 100, 400 e 4000, ognuno è divisore di quelli che lo seguono. Quindi quando si legge che "l'anno x è bisestile se e solo se x è divisibile per 4, ma non.." si deve intendere che se x è solo divisibile per 4, e non per i numeri che lo seguono, allora è bisestile.

L'anno 2004, ad esempio, è solo divisibile per 4 e non per 100, 400 o 4000, quindi, secondo il metodo di verifica che il testo illustra, il 2004 è un anno bisestile.

Quando la frase continua con "..ma non per 100,.." significa che il secondo criterio si oppone al primo, quindi se x è divisibile per 100, significa che l'anno non è bisestile (ammesso che l'anno non sia divisibile per 400 o per 4000).

Quindi gli anni 1700, 1800 e 1900, che sono divisibili per 100, ma non per 400 o per 4000, sono anni non bisestili.

La parte che segue ("a meno che") si oppone a sua volta al criterio precedente, per cui se x è divisibile per 400 (ma non per il numero che segue) l'anno è bisestile.

Quindi l'anno 2000 è un anno bisestile.[2]

Infine, se x è divisibile per 4000 ci si oppone nuovamente al criterio precedente, per cui l'anno non è bisestile.

Tenendo conto di questi ragionamenti si può modificare ulteriormente il testo per avvicinarlo ulteriormente ad un linguaggio più matematico, ottenendo:

Versione 1.03

L'anno x è bisestile **se e solo se** x è divisibile per 4 **e** non per 100, **oppure** è divisibile per 400 **e** non per 4000.

[2]Si noti che dal punto di vista strettamente storico non è corretto sostenere che l'anno 1000, ad esempio, è (era) un anno bisestile, dato che l'attuale calendario (che prevede gli anni bisestili) è stato introdotto da papa Gregorio XIII solo nel 1583, da cui il nome di calendario gregoriano.

che è molto più chiaro del testo originale (essendo anche posti in evidenza i connettivi) ed è assolutamente corretto dal punto di vista della logica matematica.

Alla prima richiesta si ritiene di aver risposto esaurientemente, avendo compreso, criticato e migliorato il testo. Si noti che stavolta si è riusciti a mantenere l'assoluta identità fra il linguaggio naturale e quello matematico, per cui le due interpretazioni coincidono.

Per affrontare efficacemente la seconda richiesta, conviene dare a ciascuna proposizione atomica un nome (solitamente si indicano con lettere minuscole dell'alfabeto inglese). Supponendo quindi di porre:

- "L'anno x è bisestile" = p;
- "x è divisibile per 4" = q;
- "x è divisibile per 100" = r;
- "x è divisibile per 400" = s;
- "x è divisibile per 4000" = t;

si ottiene una semplice traduzione del testo di versione 1.03 in simboli logici:

$$p \leftrightarrow (q \land \neg r) \lor (s \land \neg t) \tag{7.2}$$

Assegnando alla variabile x un anno qualsiasi (preferibilmente maggiore o uguale a 1583), si trasformano in proposizioni i singoli predicati e si può verificare l'espressione logica. Si noti che scrivendo la tabella di verità deve risultare che l'espressione 7.2 deve essere sempre vera, altrimenti significherebbe che il metodo per la determinazione dell'anno bisestile è errato.

Si procede, quindi, con la scrittura della tabella di verità:

q	r	s	t	$\neg r$	$\neg t$	$q \land \neg r$	$s \land \neg t$	p	$p \leftrightarrow (q \land \neg r) \lor (s \land \neg t)$
F	F	F	F	V	V	F	F	F	V
F	F	F	V	V	F	F	F	X	X
F	F	V	F	V	V	F	V	X	X
F	F	V	V	V	F	F	F	X	X
F	V	F	F	F	V	F	F	X	X
F	V	F	V	F	F	F	F	X	X
F	V	V	F	F	V	F	V	X	X
F	V	V	V	F	F	F	F	X	X
V	F	F	F	V	V	V	F	V	V
V	F	F	V	V	F	V	F	X	X
V	F	V	F	V	V	V	V	X	X
V	F	V	V	V	F	V	F	X	X
V	V	F	F	F	V	F	F	F	V
V	V	F	V	F	F	F	F	X	X
V	V	V	F	F	V	F	V	V	V
V	V	V	V	F	F	F	F	F	V

Tabella 7.1: Tabella di verità dell'espressione $p \leftrightarrow (q \land \neg r) \lor (s \land \neg t)$

Si nota che la tabella presenta molte X nelle colonne $p \leftrightarrow (q \land \neg r) \lor (s \land \neg t)$ e p. Esse sono dette **condizioni di indifferenza**. Ciò è dovuto al fatto che le 4

colonne più a sinistra della tabella formano tutte e 16 le possibili combinazioni delle 4 proposizioni atomiche q, r, s, t, mentre molte delle quali, in realtà non sono aritmeticamente possibili: ad esempio, in seconda riga la tabella di verità riporta che x è divisibile per 4000, ma non per 400, né per 100 e né per 4, il che rappresenta un assurdo aritmetico. Queste combinazioni delle proposizioni elementari non si possono mai verificare, per cui non ha senso attribuire un "risultato" alle relative proposizioni complesse. Tecnicamente, però, si preferisce elencare ugualmente tutte le possibili combinazioni delle proposizioni elementari, perché alcuni metodi di semplificazione delle espressioni finali (non considerati nelle presenti pagine) ne traggono vantaggio.

Anche la seconda richiesta è stata esaudita convenientemente.

La terza richiesta chiedeva di tracciare un diagramma di flusso che fosse indipendente dal risultato matematico ottenuto, ovvero come se l'analisi matematica non ci fosse stata. Potrebbe, però, essere interessante disegnare entrambi i diagrammi di flusso e metterli a confronto:

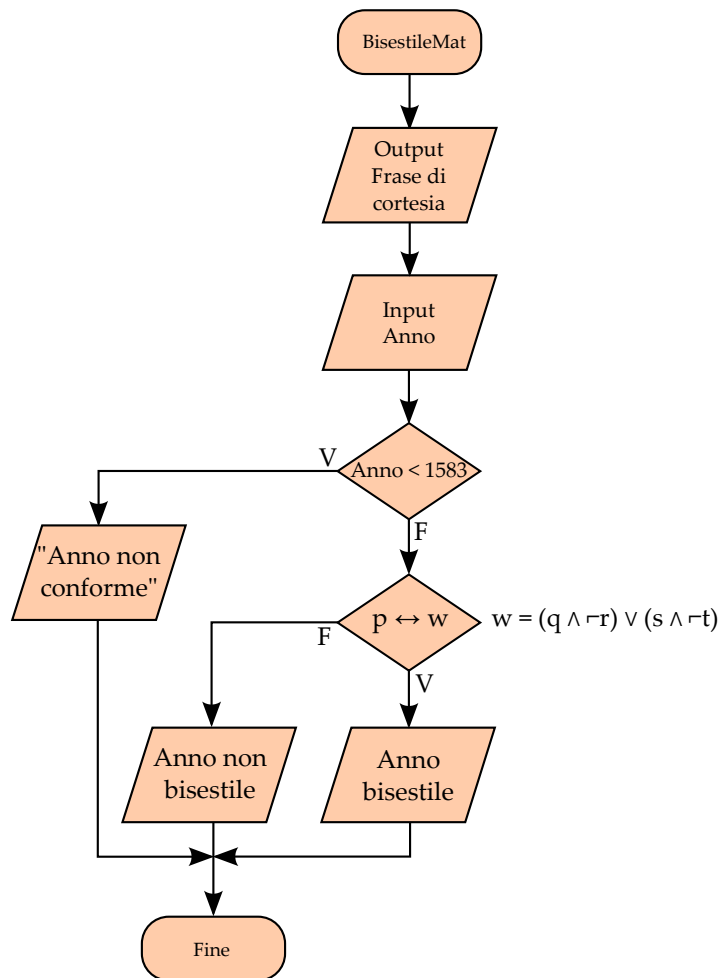



Figura 7.3: Anno bisestile "Matematico"

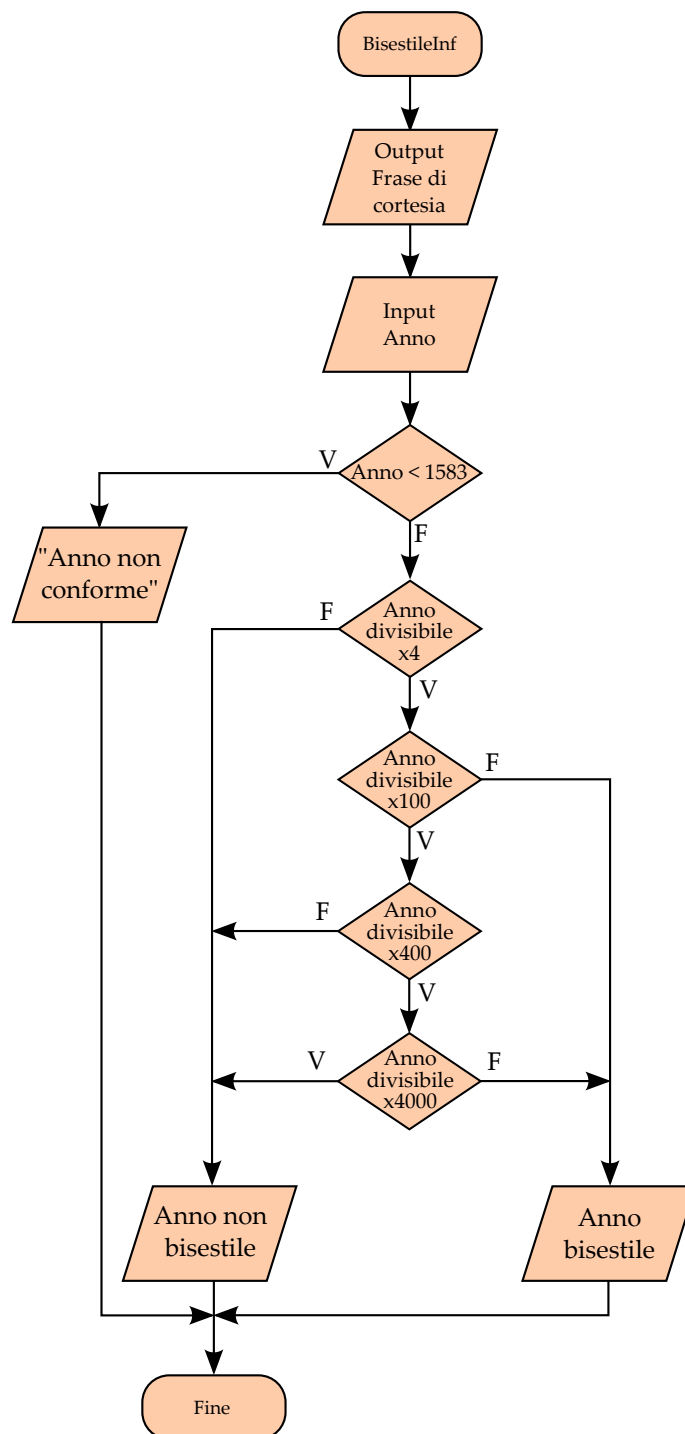



Figura 7.4: Anno bisestile "Informatico"

Il diagramma di fig. 7.3 presenta due blocchi inziali di *output/input*: nel primo blocco viene visualizzata una frase di cortesia simile a "Digitare un anno

superiore a 1583" o di ugual senso e nel secondo blocco si permette l'*input* dell'anno. Si verifica poi attraverso il blocco di selezione se l'anno è minore di 1583. In tal caso l'anno non appartiene al calendario gregoriano, per cui è privo di senso valutare se esso è stato bisestile o meno e si termina l'algoritmo. Se, invece l'anno digitato è corretto, si verifica se l'espressione 7.2 a pagina 206 è vera: se è vera, l'anno è bisestile e se è falsa non lo è. Per poter tracciare un simile diagramma di flusso bisogna aver prima determinato l'espressione 7.2 e si è visto che ciò richiede un minimo di analisi e di attenzione.

La prima parte del diagramma di fig. 7.4 a fronte è identica a quella del diagramma 7.3, per cui non si ritiene necessario alcun commento. La seconda parte è invece piuttosto diversa. Non viene testata una singola espressione logica, ma vengono testate le singole proposizioni q, r, s e t dello specchietto di pagina 206. Quest'ultimo diagramma risulta essere più complesso ed un ipotetico microprocessore eseguirebbe l'algoritmo relativo al secondo diagramma più lentamente di quanto eseguirebbe il primo.

Rimane da formalizzare il codice in linguaggio C. Si propone prima il codice del primo diagramma.

Listing 7.5: Implementazione diagramma 7.3

```
void main()
{
   int anno;    //Anno da valutare se bisestile o meno

   //Si produce un output con una frase di cortesia.
   //Il senso di tale frase e' spiegare il senso di cosa
   //si sta per fare.
   printf("Programma␣di␣valutazione␣se␣un␣anno␣digitato\n");
   printf("e'␣bisestile␣o␣meno.\n\n");
   printf("Si␣digiti␣un␣anno␣superiore␣a␣1583:␣");

   //Input dell'anno.
   scanf("%d", &anno);

   //Valutazione se detto anno e' minore di 1583. In
   //tal caso si esce dal programma e non esegue
   //alcuna valutazione sull'anno digitato.
   if (anno<1583)
   {
      printf("Anno␣non␣conforme.");
      return 0; //Esce senza fare alcuna valutazione
   }
   else
   {
      //L'anno e' OK. Valuta l'espressione logica
      //elaborata ad inizio esercizio.
      if ((anno%4==0&&anno%100!=0)||(anno%400==0&&anno%4000!=0))
         printf("L'anno␣%d␣e'␣bisestile", anno);
      else
         printf("L'anno␣%d␣non␣e'␣bisestile", anno);
      return 0; // Torna al sistema operativo
   }
}
```

Il secondo codice, come si nota facilmente, è un po' più complesso:

Listing 7.6: Implementazione diagramma 7.4

```
void main()
{
   int anno;    //Anno da valutare se bisestile o meno

   //Si produce un output con una frase di cortesia.
   //Il senso di tale frase e' spiegare il senso di cosa
   //si sta per fare.
   printf("Programma␣di␣valutazione␣se␣un␣anno␣digitato\n");
   printf("e'␣bisestile␣o␣meno.\n\n");
   printf("Si␣digiti␣un␣anno␣superiore␣a␣1583:␣");

   //Input dell'anno.
   scanf("%d", &anno);

   //Valutazione se detto anno e' minore di 1583. In
   //tal caso si esce dal programma e non esegue
   //alcuna valutazione sull'anno digitato.
   if (anno<1583)
   {
      printf("Anno␣non␣conforme.");
      return 0; //Esce senza fare alcuna valutazione
   }
   else
   {
      //L'anno e' OK. Valuta l'espressione logica
      //elaborata ad inizio esercizio predicato per predicato.
      if (anno%4==0)                    //Primo predicato
      {
         if (anno%100!=0)               //Secondo predicato
            printf("L'anno␣%d␣e'␣bisestile", anno);
         else
         {
            if (anno%400==0)            //Terzo predicato
            {
               if (anno%4000!=0)        //Quarto predicato
                  printf("L'anno␣%d␣e'␣bisestile", anno);
               else
                  printf("L'anno␣%d␣non␣e'␣bisestile", anno);
            }
            else
               printf("L'anno␣%d␣non␣e'␣bisestile", anno);
         }
      }
      else
         printf("L'anno␣%d␣non␣e'␣bisestile", anno);
      return 0; // Torna al sistema operativo
   }
}
```

Entrambi i metodi vanno bene, anche se l'approccio è piuttosto differente: un po' più teorico il primo ed un po' più pratico il secondo. E' importante una seria analisi del testo ed un approccio metodico e prudente.

7.5 L'*arithmetic if statement*

Real Programmers enjoy Arithmetic IF statements
because they make the code more interesting.
Ed Post

Si ritiene comunque doveroso dedicare una piccola sezione all'*arithmetic if statement*, nonostante sia stato messo alla berlina sia in testa alla presente sezione che nell'introduzione.

L'if aritmetico non è sicuramente da incorraggiare a tutti i costi, ma nemmeno da demonizzare. Semplicemente se ne sconsiglia l'abuso.

L'if aritmetico è l'unico *operatore ternario* del linguaggio C ed è detto anche *operatore condizionale*. Anche in virtù di tale particolarità può essere utile studiarlo con la dovuta attenzione, al fine di conoscerne pregi e difetti. L'operatore ternario accetta tre operandi, ovvero una condizione booleana e due espressioni, come evidenziato nella struttura sintattica fornita di seguito:

<condizione booleana> **?** <espressione 1> **:** <espressione 2>

dove `<espressione X>` può essere una costante, una variabile, un'espressione o una funzione e rappresenta il valore che l'operatore ritorna.

Se la condizione booleana è *non falsa* l'operatore condizionale ritorna l'espressione 1, altrimenti ritorna l'espressione 2. Ovviamente le due espressioni devono ritornare un valore appartenente allo stesso tipo o perlomeno rispettare la correlazione di tipo indicata in fig. 3.2 a pagina 126.

Il seguente esempio illustra l'uso dell'if aritmetico all'interno di un'espressione di assegnazione:

Listing 7.7: *Arithmetic if statement* in espressione di assegnazione

```
brontolo = gongolo > eolo ? cucciolo : mammolo;
```

Se `gongolo` è maggiore di `eolo`, allora a `brontolo` viene assegnato il valore di `cucciolo`, altrimenti viene assegnato il valore di `mammolo`.

L'equivalente codice utilizzante l'*if logico* è indicato di seguito:

Listing 7.8: *Logic if statement* in espressione di assegnazione

```
if (gongolo>eolo)
   brontolo = cucciolo;
else
   brontolo = mammolo;
```

I due codici sono assolutamente identici, per cui non si vede la necessità di una scrittura criptica (if aritmetico) quando si può utilizzare una scrittura sicuramente più mnemonica e di più semplice lettura (if logico).

Leggermente diversa è la seguente situazione:

Listing 7.9: *Arithmetic if statement* in struttura condizionale

```
if (gongolo>(pisolo>dotto?dotto*3:pisolo*2))
   brontolo = cucciolo;
else
   brontolo = mammolo;
```

In presenza di casi simili è evidente che l'autore delle suindicate righe di codice vuole confondere il nemico. Non c'è nessuna volontà di semplificare la lettura del codice, anzi, si rende omaggio al Vero Programmatore:

> *"Se è stato difficile scrivere il codice*
> *è giusto che sia anche difficile leggerlo."*

Siccome, però, l'autore delle presenti pagine è un programmatore scalcinato e dal cuore tenero, si propone l'analisi del codice incriminato dando i soliti valori alle variabili:

$$gongolo = 1; \quad pisolo = 2; \quad dotto = 3; \quad cucciolo = 4; \quad mammolo = 5;$$

per cui il codice diventa:

Listing 7.10: Sviluppo della struttura condizionale

```
if (1>(2>3?3*3:2*2))
   brontolo = 4;
else
   brontolo = 5;


if (1>4)
   brontolo = 4;
else
   brontolo = 5;
```

per cui `brontolo = 5`.

Naturalmente il codice presentato era solamente un esempio (da non seguire!) ma potrebbe essere illuminante: si può effettivamente scrivere del codice che può essere difficilmente comprensibile ai più se non ci si vota alla semplicità. Una scrittura semplice e piana favorisce, invece, sia la lettura che la ricerca degli errori di programmazione, per cui non si vede la necessità di scrivere del codice di non immediata comprensione, anche se questo comporta qualche riga di codice in più.

7.6 La scelta multipla

Una terza struttura decisionale fornita dal linguaggio C è la *scelta multipla*. Si tratta di una struttura decisionale che non analizza una condizione booleana ma un'espressione intera che può assumere diversi valori scalari.

L'istruzione che identifica la struttura di scelta multipla è lo `switch` e ha la seguente sintassi:

Listing 7.11: Data di nascita dei Beatles

```
switch(beatle)
{
   case kJohn:  anno = 1940;    break;
   case kPaul:  anno = 1941;    break;
   case kGeorge:anno = 1943;    break;
   case kRingo: anno = 1940;
}
```

In figura 7.5 sono evidenziate le parti fondamentali della struttura multidecisionale.

L'operando dello `switch` deve essere un intero, ovvero uno scalare. Non si può quindi usare come argomento dello `switch` un numero in virgola mobile, perché il tipo non è scalare. Ciò si rende necessarrio perché deve essere possibile poter determinare, dato un valore dell'argomento, quale sia il valore *precedente* e quale quello *successivo*.

espressione scalare

```
switch(beatle)
{
    case kJohn:   anno = 1940; break;
    case kPaul:   anno = 1941; break;
    case kGeorge: anno = 1943; break;
    case kRingo:  anno = 1940;
}
```

costanti

break finali

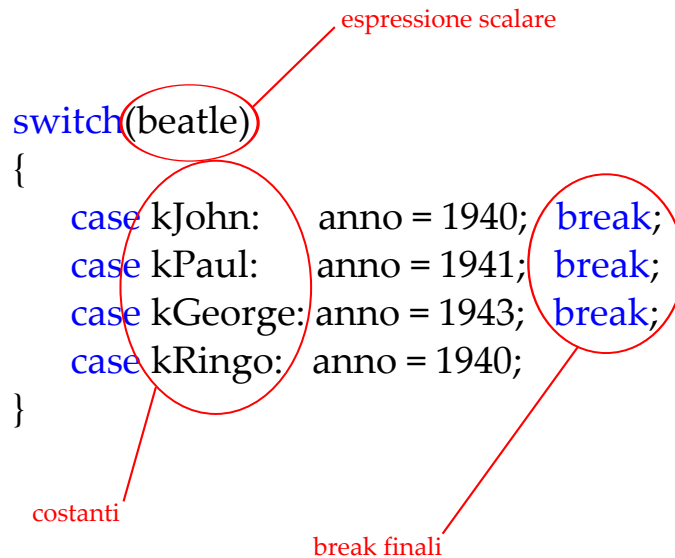



Figura 7.5: Struttura dello switch

Alla sinistra della parola chiave `case` deve essere posta *una costante oppure un'espressione di costanti*, seguita dai due punti (simbolo ":"). **Non è possibile utilizzare una variabile o un'espressione o una funzione che restituisce una variabile.** L'allievo ponga molta attenzione a ciò, perché l'uso di variabili nel `case` è un errore frequentissimo.

Lo `switch` valuta il valore dell'argomento scalare e lo **confronta** con quello indicato nei vari `case`. Quando e se lo trova, esegue l'espressione che segue il `case`. Se, ad esempio, la variabile `beatle` dovesse assumere il valore `kGeorge`, verrebbe eseguita l'assegnazione `anno = 1943;` e poi il `break`. Quest'ultima istruzione **forza l'uscita** dallo `switch`. Il ruolo del `break` risulta quindi essere fondamentale. Si veda a tal proposito il seguente esempio:

Listing 7.12: Esempio di switch senza break

```
switch(x)            //Si supponga x = 5.
{
   case 4: x += 3;
   case 5: x += 7;   //Viene eseguita questa espressione...
   case 6: x += 11;  //...ma anche questa...
   case 7: x += 23;  //...e questa.
}
```

Si supponga che la variabile x assuma inizialmente il valore 5. Lo `switch` riconosce che il secondo `case` indica l'esatto valore della variabile e viene ese-

guita l'espressione correlata al secondo `case`, ossia `x += 7;`. Si noti, però che ciò non significa assolutamente che terminata l'esecuzione di detta espressione si possa uscire dallo `switch`, anzi, si prosegue fino al primo `break` o fino alla fine dell'istruzione, ovvero dopo l'ultimo `case`. Quindi nel presente esempio, al valore di x viene prima sommato il valore 7, poi il valore 11 ed infine il valore 23, per cui si otterrà $x = 46$. La prima espressione non viene eseguita perché l'*entry point* è determinato dall'uguaglianza fra valore dell'argomento dello `switch` e la costante del `case`.

Se la varibile x avesse assunto il valore 3, si noti che **nessuna delle espressioni presenti sarebbe stata eseguita**. Ciò ci introduce ad una ulteriore possibilità, ovvero all'uso dello *statement* `default`, come evidenziato nel seguente codice:

Listing 7.13: Esempio di switch con default

```
switch(x)                //Si supponga x = 3.
{
   case 4: x += 3;  break;
   case 5: x += 7;  break;
   case 6: x += 11; break;
   case 7: x += 23; break;
   default: x = 0;       //Viene eseguita questa espressione
}
```

Se il valore espresso dall'argomento non trova alcun *matching* fra i valori indicati dai `case`, viene automaticamente eseguita l'espressione indicata dallo *statement* `default`.

Questa variante dello `switch` può risultare utile se la variabile x deve sempre e comunque assumere un determminato valore. Nel caso presentato dal codice 7.13 non sarebbe possibile ottenere detto risultato senza il `default`, dato che la variabile che viene modificata nelle espressioni associate ai `case` è la stessa che viene valutata dallo `switch`. Se, ad esempio, la variabile testata dallo `switch` fosse stata diversa, si sarebbe potuto assegnare alla variabile che viene modificata nel corpo dello `switch` un valore di inizializzazione prima della scelta multipla. Nel presente esempio, però, tale strategia non è applicabile.

Si fa notare che tutte le espressioni del corpo dello `switch` sono simili fra loro, nel senso che modificano la variabile x aggiungendo ad essa un determinato valore in dipendenza del valore originale di detta variabile. Le espressioni potrebbero, però essere completamente differenti, come nel presente esempio:

Listing 7.14: Altro esempio di switch con default

```
switch(x)
{
   case 4: x += 3;                      break;
   case 5: pluto = 34;                  break;
   case 6: printf("Terza␣espressione"); break;
   case 7: scanf("%d", pippo);          break;
   default: x++;
}
```

Si noti che ciascun `case` può ospitare anche più istruzioni e che *non si rendono necessarie le parentesi graffe*, che comunque non sono errate. Il seguente codice è quindi valido in ogni sua parte:

Listing 7.15: Esempio di switch con graffe e senza

```
switch(x)
{
   case 4: {x += 3; y++;}        break; //Con graffe
   case 5: pluto = 4; pippo = 3; break; //Senza graffe
   case 6:
   {                                    //Con graffe
      x += y;
      break;
   }
}
```

La struttura multidecisionale illustrata attraverso il codice 7.13 è rappresentabile graficamente mediante il diagramma di flusso illustrato in fig. 7.6.

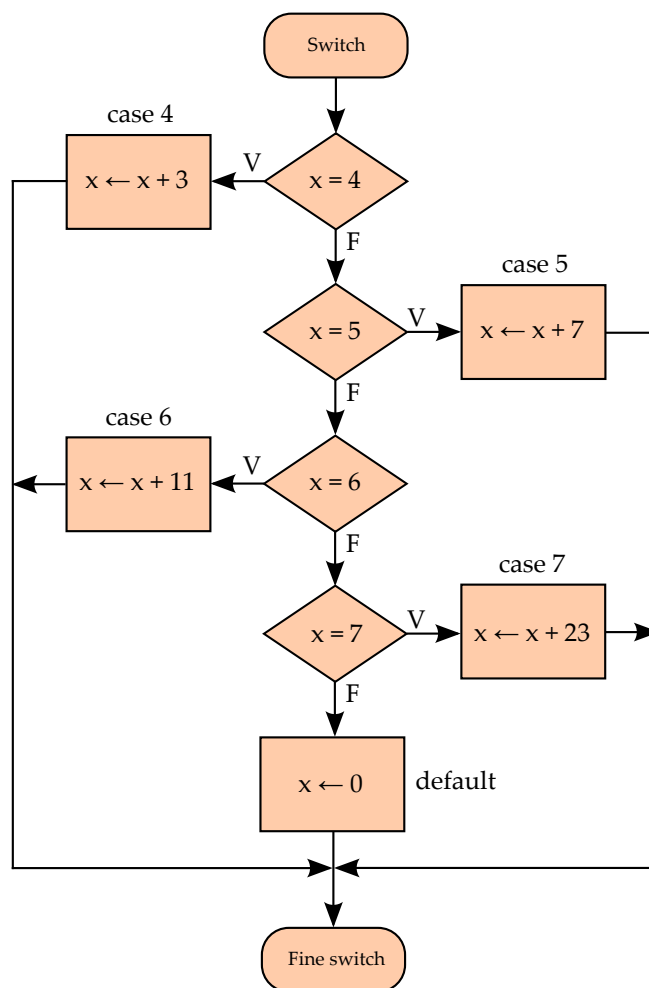



Figura 7.6: Esempio di switch con default

Naturalmente tale struttura è facilmente implementabile attraverso una se-

rie di `if..else` posti in cascata fra loro. Una siffatta pratica non è però consigliabile, perché più prolissa e meno immediata rispetto alla scelta multipla.

7.7 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 7.

Capitolo 8

Le iterazioni

Le iterazioni costituiscono la terza struttura fondamentale, insieme alla sequenza e alla selezione, della programmazione. Esse permettono l'esecuzione ciclica di una sequenza, ovvero la *ripetizione di una sequenza per un certo numero, definito o meno, di volte*. Le iterazioni possono essere definite attraverso due caratteristiche fondamentali:

- il tipo di test;
- la quantificazione dei cicli.

Le iterazioni possono essere classificate, infatti, a *test iniziale* o a *test finale* e a numero di cicli *noto a priori* oppure no. Questi due tipi di classificazione permettono di raggruppare le iterazioni in tre categorie distinte:

- iterazioni con numero di cicli noto a priori;
- iterazioni con numero di cicli non noto a priori e a scelta iniziale;
- iterazioni con numero di cicli non noto a priori e a scelta finale.

Questa classificazione apre una discussione annosa, il cui argomento centrale lascia solitamente piuttosto perplesso lo studente: perché tre tipi di iterazioni diverse? La domanda è pertinente e lo dimostra il fatto che regni una certa confusione sull'argomento, sia che si cerchi una plausibile risposta sul Web che in biblioteca. Si cercherà di rispondere nella maniera più chiara e più convincente possibile.

8.1 Perché tre iterazioni diverse?

Innanzi tutto è bene chiarire subito che si potrebbe benissimo utilizzare un solo tipo di iterazione per risolvere dignitosamente qualsiasi tipo di problema informatico legato ai cicli. Ma allora perché tre tipi di cilci diversi? Moltissime fonti propongono risposte anche assai diverse fra loro. Quella che le presenti pagine vogliono proporre è di natura storico-logica.

8.1.1 La risposta storica...

La programmazione strutturata è nata come esigenza metodologica per cercare di minimizzare gli errori semantici di programmazione, i cosidetti "bachi" (traduzione onomatopeica di *bugs*). Negli anni '60-'70 si è cercato quindi di "concettualizzare" la programmazione, *tipizzando* le variabili, "abolendo" il `goto` e introducendo strutture sintattiche più evolute e più rigide al fine di limitare la sfrenata fantasia del programmatore.

Come già accennato nel precedente capitolo, nella seconda metà degli anni '60, Böhm e Jacopini, formularono il *Teorema di separazione*, mediante il quale si dimostrò che era possibile scrivere un qualsiasi programma senza l'uso dei `goto`, ovvero che un qualsiasi programma poteva essere formato mediante le sole strutture di controllo della *sequenza*, della *selezione* e dell'*iterazione*.

In particolare, le iterazioni rispondono a molteplici esigenze anche piuttosto differenti fra loro, ma tutte raggruppabili in tre grandi categorie[1] e risolvibili mediante i cicli identificati nella pagina precedente.

Le tre diverse iterazioni rispondono a questo tipo di logica, ovvero ad un ordine "imposto" attraverso strutture sintattiche piuttosto rigide ciascuna delle quali identificanti una ben precisa categoria di problemi.

8.1.2 ...e quella logica

Le suddette categorie di problemi corrispondono alle seguenti iterazioni:

iterazioni con numero di cicli noto a priori - Questo ciclo va usato[2] quando si sa a priori, ovvero prima di entrare per la prima volta nel corpo del ciclo, quante volte si dovrà iterare, cioè eseguire la sequenza racchiusa nel ciclo. Va specificato meglio cosa si intenda per "noto a priori". Si intende con ciò che il numero delle iterazioni deve essere *conosciuto oppure calcolabile con esattezza* prima del ciclo;

iterazioni con numero di cicli non noto a priori e a scelta iniziale - Questo ciclo va usato quando *non* è noto il numero delle iterazioni a priori *ed esiste* la possibilità di non entrare mai nel ciclo;

iterazioni con numero di cicli non noto a priori e a scelta finale - Questo ciclo va usato quando *non* è noto il numero delle iterazioni a priori *e si deve* entrare nel ciclo almeno una volta.

Tutto ciò significa che quando si rileva la presenza di un ciclo in un problema dato, la prima cosa da fare dovrebe essere *identificare a quale delle tre categorie appartiene*. Questo è di solito un esercizio mentale che lo studente fa mal volentieri, limitandosi ad usare il ciclo che conosce meglio. Una tale mentalità porta, però, con sé numerosi effetti collaterali: non si impara il linguaggio C come si dovrebbe; non si abitua il cervello all'astrazione; si rimane perennemente in superficie al problema restando incapaci di effettuare un'analisi profonda, ecc.

[1]Böhm e Jacopini parlarono nel loro teorema del solo ciclo while che risulta essere sufficiente per risolvere un qualsiasi tipo di problema legato ai cicli.

[2]Ovviamente non c'è nessun *obbligo* formale ad usare un determinato tipo di ciclo in base al problema presentato. Si consiglia, però, molto vivamente di attenersi a tale regola per abituare la mente al ragionamento logico e per allenarsi a riconoscere le diverse situazioni del mondo reale e ricondurle a strutture sintattiche codificate.

Siccome il metodo serve anche a guidare i primi passi, si propone il seguente schema da usare per identificare il tipo di ciclo dato un determinato problema:

1. **Step 1**. *E' noto oppure calcolabile il numero di volte che la sequenza racchiusa nel ciclo deve essere eseguita?* Se la risposta è "Si" e se si può rispondere affermativamente *prima* di eseguire il ciclo, allora è stata identificata l'**iterazione con numero di cicli noto a priori**. Altrimenti si deve passare allo *step* 2. Esempi reali di siffatte situazioni sono i seguenti:
   - Pensare due volte prima di aprir bocca.
   - Masticare tre volte a destra e tre a sinistra.

   In entrambi i casi i numeri citati non sono vincolanti (si potrebbe riformulare le due frasi con quattro, cinque, sei, ecc.), però sono noti.
2. **Step 2**. *Non essendo noto il numero di cicli, si deve eseguire la sequenza racchiusa nel ciclo almeno una volta?* Se la risposta è "Si" allora si è identificata l'**iterazione a test finale**, altrimenti si deve passare allo *step* 3. Esempi reali di siffatte situazioni sono i seguenti:
   - Impastare finché la farina si amalgama con l'uovo.
   - Lasciar squillare il telefono in attesa di risposta.

   Quando si aggiunge l'uovo alla farina, sarà sempre non amalgamato ad essa, nè si può pretendere risposta al telefono se non lo si fa squillare almeno una volta.
3. **Step 3**. *Non essendo noto il numero di cicli, esiste la possibilità che la sequenza racchiusa nel ciclo non debba essere eseguita nemmeno una volta?* Se la risposta è "Si" allora si è identificata l'**iterazione a test iniziale**, altrimenti si è commesso qualche errore e si deve ritornare allo *step* 1. Esempi reali di siffatte situazioni sono i seguenti:
   - Aggiungere zollette di zucchero a piacimento nel caffè.
   - Spalare la neve finché il vialetto diventa sgombro.

   Chi preferisce il caffè amaro non aggiungerà nemmeno una zolletta, come pure non è detto che il vialetto sia innevato.

Rispondere alle suddette tre domande in maniera critica e non superficiale permette di affrontare la scelta del tipo di iterazione da usare con metodo e ordine e permette anche di minimizzare gli errori durante la scelta.

Fatte queste doverose premesse si può analizzare la prima delle iterazioni: quella a numero di cicli noto a priori.

8.2 L'iterazione con numero di cicli noto (ciclo *for*)

L'iterazione con numero di cicli noto è una struttura di relativa complessità. Essa utilizza un indice o un contatore in incremento o in decremento per contare il numero delle iterazioni da eseguire. L'indice/contatore deve essere inizializzato *prima* dell'esecuzione del primo ciclo e deve essere aggiornato in incremento o in decremento *dopo ogni* esecuzione della sequenza. Il valore dell'indice/contatore deve essere valutato *prima* dell'esecuzione di ogni sequenza. Per tale motivo l'iterazione con numero di cicli noto a priori è un'iterazione a test

iniziale, per cui la sequenza potrebbe non essere mai eseguita nemmeno una volta. Una possibile rappresentazione grafica mediante diagramma di flusso di un'iterazione a numero di cili noto a priori con indice in decremento è data in fig. 8.1.

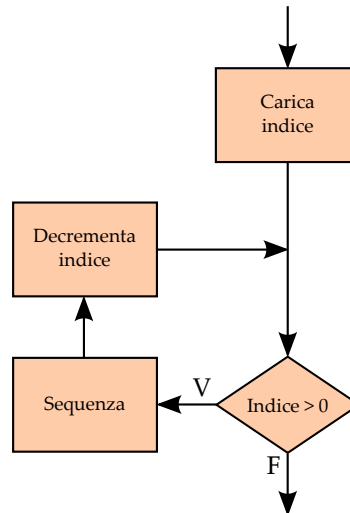



Figura 8.1: Iterazione con numero di cicli definito

La struttura sintattica dell'iterazione con numero di cicli definito è nota, nel linguaggio C, come *ciclo for* ed ha la seguente forma:

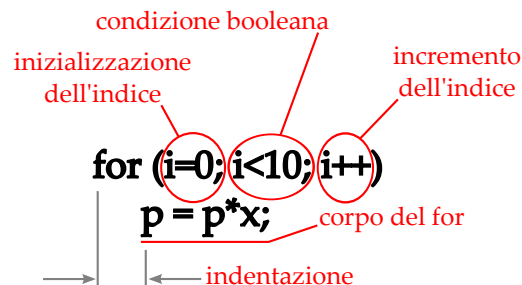



Figura 8.2: Struttura del ciclo *for*

Prima di commentare la struttura sintattica del ciclo `for`, è venuto il momento di spiegare attraverso un esempio la differenza fra *espressione* e *istruzione*. La fig. 8.2 illustra l'**istruzione** relativa al ciclo `for` e nella prima riga della struttura sintattica sono evidenziate delle **espressioni** separate da punto e virgola.

Questo ci porta a formulare l'ipotesi che l'espressione sia solo una *componente* dell'istruzione, anche se spesso l'unica.

In effetti, la prima delle espressioni è l'*espressione di inizializzazione dell'indice del ciclo*. Nel presente caso l'indice è la variabile intera `i` che viene inizializzata al valore 0. La seconda espressione è la *condizione booleana che, se non falsa, permette l'esecuzione del corpo del ciclo*. Si noti che essa viene eseguita prima

dell'eventuale esecuzione del corpo del ciclo, il che significa che il ciclo `for` è un ciclo a test iniziale e che quindi si potrebbe non entrare mai nel corpo del ciclo. La terza espressione è l'*aggiornamento dell'indice*.

E' piuttosto importante notare che le tre espressioni vengono eseguite *automaticamente dal for*, senza la necessità che vengano richiamate dal programmatore ed è anche importante capire quando le suddette espressioni vengono eseguite. L'espressione di inizializzazione viene eseguita *una tantum* quale prima azione dell'istruzione. L'espressione booleana viene valutata subito dopo e *prima di ogni eventuale esecuzione della sequenza contenuta nel corpo del ciclo*. Se la condizione booleana è *non falsa*, si entra nel corpo del ciclo, altrimenti si termina l'esecuzione del ciclo `for` e si prosegue con le istruzioni successive che lo seguono.

Se si entra nel corpo del ciclo, *prima* si esegue tutta la sequenza formante il corpo del ciclo e *dopo* si esegue l'espressione di aggiornamento dell'indice[3].

8.2.1 Un errore frequente nell'uso del *for*

A tal proposito si vuole richiamare l'attenzione su un frequente errore compiuto dagli studenti, che è assolutamente orribile dal punto di vista concettuale: l'aggiornamento dell'indice eseguito direttamente dal programmatore nel corpo del ciclo. Si veda a titolo d'esempio il sottostante codice:

Listing 8.1: Pessimo esempio di codice

```
for(i=0; i<10; i++)
{
   //Codice simile non deve essere mai scritto!!
   //Se si sente il bisogno di forzare l'uscita
   //dal ciclo for significa che si e' sbagliata
   //la scelta del ciclo!
   if(p>1000)   //Se p e' maggiore di 1000...
      i = 10;   //...si forza l'uscita dal ciclo
   else
      p = p*x;
}
```

Nel corpo del ciclo si modifica il valore dell'indice `i`, portandolo a 10, al fine di forzare l'uscita dal ciclo nel caso in cui la variabile `p` dovesse superare un determinato valore.

Scritture simili vanno assolutamente evitate, perché totalmente in contrasto con la programmazione strutturata, ovvero con lo spirito stesso che giustifica i linguaggi ad alto livello come il C.

A quel programmatore che però volesse ignorare le suddette suppliche dedico il seguente pensiero:

> "Sciupatore di carta[4], di tempo e di cervelli." ☺
>
> **L'arte di insultare** - Arthur Schopenhauer

[3]Si noti che l'aggiornamento dell'indice viene sempre eseguito *dopo* che è stato completamente eseguito il corpo del ciclo, anche se l'espressione di aggiornamento dell'indice dovessere essere **++i** anziché **i++**.

[4]Inteso come supporto di scrittura: non esistevano ancora i PC ai tempi di Schopenhauer.

Sperando di aver convinto il maggior numero di studenti possibile a non modificare mai l'indice nel corpo del `for`, si possono illustrare altre particolarità di detta istruzione.

8.2.2 Sintassi strane, permesse e sconsigliate

Una particolarità del ciclo `for` riguarda il numero delle espressioni racchiuse tra parentesi tonde. Si è maliziosamente lasciato intuire che le espressioni dell'istruzione `for` siano tre, ma ciò è falso. Sono semplicemente separate da punto e virgola, esattamente come è stato prudentemente fin qui detto.

Quindi sembra non essere così importante che le espressioni racchiuse fra le parentesi tonde del `for` siano tre. In effetti, sono assolutamente corrette le espressioni sintattiche di seguito fornite:

Listing 8.2: Espressioni sintatticamente consentite nel ciclo *for*

```
//Esempio 1: permesso
for(i=0, pippo=0; i<10; i++)
   pippo += i*2;


//Esempio 2: sconsigliato
for(i=0, pippo=5.3, pluto = 0; i<10, pippo<10; i++, pippo+=2.1)
   pluto += pippo*2;


//Esempio 3: strano, ma usatissimo
for(;;)
{
   //Si supponga che il corpo del for
   //sia relativamente complesso e
   //rappresenti il ciclo principale di
   //un piccolo sistema embedded.
}
```

Accertata la correttezza sintattica dei suddetti tre esempi, esaminiamoli dal punto di vista semantico.

L'esempio 1 riporta un'espressione di inizializzazione leggermente differente rispetto a quella indicata in fig. 8.2 a pagina 222. In particolare le espressioni sono due e separate da virgola. Trattandosi dell'espressione di inizializzazione, significa che sia la variabile `i` che `pippo` verranno inizializzate *una tantum* rispettivamente al valore 0 e 5, prima che venga valutata per la prima volta l'espressione booleana.

Ciò è lecito e consigliabile per quelle variabili che trovano totale o principale significato all'interno del corpo del `for` e non sono utilizzate fuori da esso se non marginalmente, ad esempio, per valutare il risultato prodotto dal ciclo.

Il secondo codice è un chiaro esempio di crittografia paleoinformatica. Il fine non è scrivere codice efficiente, ma confondere il nemico. Ciò nonostante si vuole ugualmente analizzare detto codice anche se non si vuole assolutamente incorraggiarne la scrittura e la diffusione.

L'inizializzazione viene eseguita per tre variabili: `i`, `pippo` (addirittura in virgola mobile) e `pluto`. Si può restare perplessi dal punto di vista stilistico, ma si è visto di peggio.

La condizione booleana vede due espressioni separate da virgola. Come vanno intese? Si deve immaginare che siano logicamente unite dalla congiunzione logica, quindi affinché venga eseguito il corpo del `for` è necessario che *entrambe* le espressioni logiche siano vere.

L'aggiornamento dell'indice vede anche l'aggiornamento di `pippo`. Anche in questo caso non si tratta di nulla di troppo strano. Anzi, si coglie l'occasione per far notare che l'aggiornamento può essere anche diverso dal semplice incremento o decremento della variabile. Nel presente caso, `pippo` viene aggiornato sommando 2 al suo valore precedente.

Preso nel complesso, però, il secondo esempio è da sconsigliare perché criptico, scarsamente leggibile e si obbliga il lettore a ragionamenti, magari anche fantasiosi e creativi, ma poco lineari.

Il terzo codice è usatissimo da chi scrive software su microcontrollori per sistemi *embedded*. Il `for` è semplicemente privo di espressione di inizializzazione, di condizione booleana e di aggiornamento dell'indice, per cui il risultato è un *loop* infinito dal quale non si esce mai. Ciò è quanto si vuole solitamente ottenere dopo aver eseguito l'inizializzazione del sistema.

8.3 Esercizi svolti con il ciclo *for*

I seguenti esempi vogliono solamente illustrare il ciclo `for` e non ancora rappresentare piccoli programmi autonomi. Ci si concentrerà, quindi, solamente sul ciclo `for`, non facendoci distrarre da altre strutture.

Esercizio - ◊◊◊ Una semplice serie

Sia data la seguente serie, indicata in 8.1, convergente a 2:

$$1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + ... + \frac{1}{2^{n-1}} \qquad (8.1)$$

Si vuole calcolare il valore che detta serie assume con 20 termini.

Soluzione

Si può immaginare la 8.1 riscritta nel seguente modo:

$$\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + ... + \frac{1}{2^{n-1}} \qquad (8.2)$$

con $n = 20$.

Ogni termine implica, ad una prima e sommaria analisi, gli stessi calcoli, il che significa che i singoli termini possono essere calcolati mediante un ciclo che ripete sempre le stesse operazioni algebriche.

Una volta determinata la presenza di un ciclo ci si deve chiedere se il numero di iterazioni è noto oppure no. Siccome i termini richiesti sono 20 significa che siamo in presenza di un'iterazione a numero di cicli noto a priori e siamo in grado di identificare la necessità di un ciclo `for`.

Prima di avventurarci immediatamente nella scrittura del ciclo è bene provvedere a tracciare un diagramma di flusso, che permetta di riflettere sull'algoritmo mentre se ne traccia una grossolana soluzione. Sicuramente si può usare come base di partenza la struttura grafica della figura 8.1 a pagina 222.

Un possibile diagramma di flusso relativo all'algoritmo di risoluzione dell'esercizio potrebbe essere quello di fig. 8.3. Il primo blocco evidenzia l'inizializzazione delle variabili, che si suppongono intere (`i` e `p`) e di tipo `float` (`t`). Esse sono l'indice `i`, il termine `t` e la potenza `p`. L'indice deve essere inizializzato a 0, come pure `t`, mentre la potenza `p`, ovvero il denominatore dei singoli termini, deve essere inizializzato a 1. Il blocco rappresentante la sequenza, ovvero le istruzioni contenute nel corpo del ciclo, devono essere eseguite e lette dall'alto al basso. Quindi deve essere prima eseguito il calcolo di t e poi quello di p. Chiariti questi dettagli è possibile passare subito dal diagramma di flusso

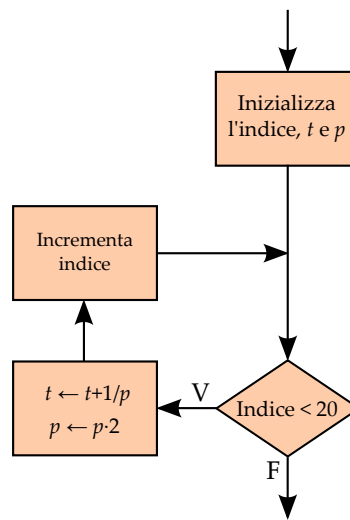



Figura 8.3: Algoritmo di calcolo della serie

al codice:

Listing 8.3: Calcolo della serie con ciclo *for*

```
for (i=0, t=0.0, p=1; i<20; i++)
{
   t += 1.0/p;   //Si noti il numeratore
   p *= 2;       //Aggiorna il denominatore
}
```

Si nota un'espressione di inizializzazione piuttosto importante, mentre la condizione booleana e l'aggiornamento dell'indice sono del tutto ortodossi.

Si sarebbe potuto, però, scrivere il ciclo anche nella seguente maniera:

Listing 8.4: Calcolo alternativo della serie con ciclo *for*

```
for (i=0, t=0.0, p=1; i<20; p *=2, i++)
   t += 1.0/p;   //Si noti il numeratore
```

Così facendo il corpo del `for` si riduce all'essenziale, mentre l'aggiornamento della variabile relativa alla potenza viene spostata nell'espressione di aggiornamento dell'indice.

I due codici sono assolutamente identici, anche se si ritiene preferibile il primo. Un'ulteriore variante potrebbe consistere nell'inizializzare le variabili `t` e

`p` prima del ciclo, dedicando le tre espressioni del `for` completamente all'indice. Quest'ultima soluzione è da preferirsi e sarà quella attuata nel prossimo esercizio.

Esercizio - ◊◊◊ Calcola la media

Si supponga di voler calcolare la media aritmetica di 10 numeri interi digitati da tastiera. Si chiede la sola scrittura del codice dell'eventuale ciclo (e di qualche eventuale istruzione ad esso collegata), naturalmente corredato di apposito diagramma di flusso.

Soluzione

La presenza di un ciclo sembra evidente: si devono ripetere per 10 volte le stesse azioni. E' possibile che dette azioni non siano ancora chiare nella mente dell'allievo, ma sembra altamente probabile la presenza di un ciclo.

Il passo successivo consiste, quindi nel determinare quale ciclo sia più appropriato per attuare una possibile soluzione. Utilizzando la procedura schematizzata a pagina 221, ci si deve quindi chiedere se il numero di cicli è noto. La risposta è sicuramente "Si", per cui si può optare per un'iterazione con numero di cicli definito a priori, ossia un ciclo `for`.

Si può ora cercare di definire il diagramma di flusso. Per poter calcolare la media aritmetica di 10 numeri digitati da tastiera è sufficiente sommarli tutti e dividere per 10 il risultato ottenuto. Il diagramma potrebbe, quindi, essere simile al presente:

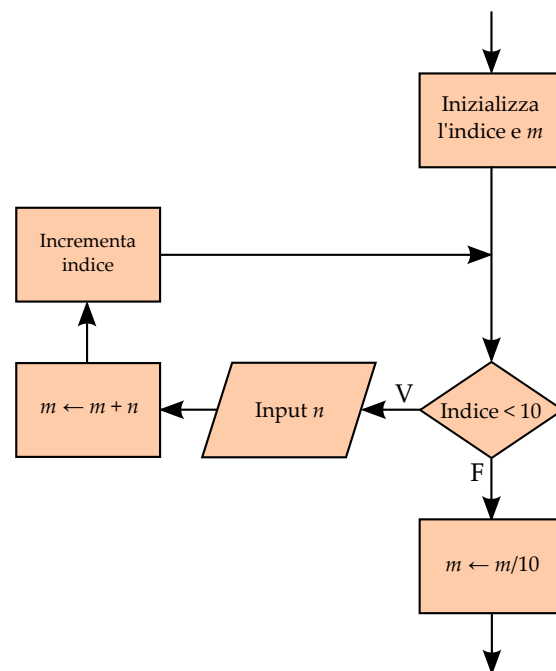



Figura 8.4: Algoritmo di calcolo della media

Oltre all'indice si fa uso di ulteriori due variabili: `n` atta a contenere il singolo numero digitato da tastiera e `m` atta a contenere la media. Quest'ultima

variabile serve anche a contenere temporaneamente la somma di tutti i numeri n digitati da tastiera. Detta somma dovrà essere divisa per 10 appena finito il ciclo.

Diversamente da quanto fatto nell'esercizio precedente si decide di inizializzare la variabile m fuori dal ciclo e di dedicare le tre espressioni del `for` al solo indice, come buona prassi consiglia.

Si noti che non vi è alcun bisogno di inizializzare la variabile n (vedi sez. 3.4) prima dell'*input*.

Fatti questi brevi commenti si può senz'altro passare al codice riferito al diagramma di fig. 8.4.

Listing 8.5: Calcolo della media con ciclo *for*

```
m = 0.0;                //Azzera la somma parziale e la media
for (i=0; i<10; i++)
{
   scanf("%d", &n);     //Digita il numero iesimo
   m += n;              //Somma il numero digitato alla
                        //somma parziale
}

//Calcola la media aritmetica dei numeri digitati. La
//variabile m e' il risultato.
m /= 10;
```

Si noti il maggior ordine sintattico del `for` rispetto all'esercizio precedente. Il codice è meno "compatto", ma più leggibile.

Esercizio - ◊◊◊ Trova il minimo

Si chiede la stesura del diagramma di flusso e del corrispondente codice in linguaggio C relativo alle seguenti azioni:

1. digitare 10 numeri relativi;
2. trovare il minore dei 10 numeri;
3. visualizzare a video il numero trovato.

Soluzione

Un problema del tutto simile è già stato trattato altrove, mediante la sola stesura del diagramma di flusso.

Sicuramente si può ipotizzare anche in questo caso la presenza di un ciclo, dato che la richiesta 1 chiede esplicitamente di ripetere la stessa azione per 10 volte. Anche l'azione indicata al punto 2 lascia trasparire, anche se in maniera un po' più oscura, un'azione che si ripete, ovvero la ricerca. L'ultima richiesta sembra, invece, non far parte di alcun ciclo.

Anche in questo caso l'identificazione del tipo di ciclo non sembra creare problemi, dato che si esplica chiaramente che le prime due suddette azioni vanno eseguite per 10 volte. Si tratterà, anche in questo caso, di un ciclo `for`[5].

La difficoltà che solitamente l'allievo incontra è posta altrove. Di solito si nota una certa difficoltà nel definire l'algoritmo da eseguire, piuttosto che le istruzioni da scrivere. Conviene quindi fare una riflessione metodologica

[5] Vi erano altri sottilissimi indizi, di cui non si è tenuto conto, che indicavano la presenza di un ciclo *for*: ad esempio, il fatto che la presente sezione sia tutta dedicata al ciclo *for*.

sull'argomento, cercando di rispondere alla seguente domanda: come si fa a "pensare" una possibile soluzione?

Sembra che dietro la ricerca della soluzione ci sia un processo creativo riservato a poche persone particolarmente intelligenti e che di solito portano "degli occhiali da sole per avere più carisma e sintomatico mistero" (cit.).

L'iconografia mitologica dello studente "genio" serve di solito da scudo allo studente normodotato, che sembra dica: "La soluzione di problemi così complessi è affare da geni, non da persone normali come me". Se, però, si ipotizza che la soluzione del suddetto problema sia affare da persone normali, per la proprietà transitiva, lo studente di cui sopra potrebbe sentirsi di intelligenza inferiore alla media. Nulla di più sbagliato. Basta un po' di metodo e un po' di allenamento.

Chi ha difficoltà a tracciare un algoritmo non dovrebbe pensare a diagrammi di flusso, che non sono un mezzo per ideare algoritmi, ma solo per rappresentarli, o a programmi in linguaggio C, che si scrivono sempre *dopo* che è stato elaborato un algoritmo, ma farebbe bene a *prendere carta e penna*.

Un buon inizio per ideare un algoritmo è partire da *situazioni il meno complesse possibile*, cercando di evitare situazioni particolari; cercare di "estrarre" un insieme di regole o un modello che permetta di arrivare alla soluzione e poi estendere le regole o il modello a situazioni via via più complesse.

Si potrebbe, ad esempio, supporre che i numeri di cui si deve trovare il minimo siano solamente tre. Ad esempio

3 -5 19

Qui nasce un problema. La sfida che abbiamo proposto al nostro cervello è troppo banale: ci ritroviamo un cervello troppo potente per sfruttare effettivamente le sue capacità. Il nostro cervello "bypassa" tutti i ragionamenti ed estrae immediatamente senza apparente difficoltà il numero -5 come minore dei tre numeri. Probabilmente qualche studente potrebbe avere difficoltà a dire come è riuscito a trovare detto numero. Effettivamente se il problema che proponiamo al nostro cervello è troppo banale, "fa tutto lui" e non riusciamo nemmeno a dire che ragionamenti abbiamo fatto.

Supponiamo, però, la seguente sequenza di numeri:

33 456 962 994736 187 -34 -6789 452 6 0 45 -78234 321 -349 7 650440 887733
63628 -5463911-630 85103 44 -100029 -815029 27100915 2 1 6017329 882 -6101935
-20117 -1 392 9994815 -99171 -193729 -816204 - 1213472 -9163028 -1529273 81028
-891263 91627 15203 8811663 -710294 -8173029 -9155287 -100000 -201839 -19827
-5916 9173 251 -61 -91982 -884734 -9554543 756 9556 -283645 -37365 -78 -5543
207976 -7664 7687 -1323 -967648 -655343 -7676443 885 654543 68658 -98685 6291

E' abbastanza evidente che il "colpo d'occhio" non aiuta molto ed è necessario utilizzare un metodo. Magari un metodo tale che risulti essere utile per qualsiasi lista di numeri, lunga e complessa a piacere. Adesso è sicuramente necessario attuare una qualche strategia.

La domanda da porsi diventa ora: "Se la lista di numeri fosse lunghissima, che strategia si potrebbe adottare per trovare il numero minore, utilizzando solo carta e penna?"

Probabilmente a questo punto anche lo studente meno fantasioso potrebbe incominciare ad avanzare qualche timida proposta. Si potrebbe, ad esempio, scrivere sul foglio di carta il primo numero e poi confrontarlo con il secondo, col terzo, col quarto, ecc. finché si trova un numero minore di quello scritto. Una volta trovato si "aggiorna" il minore fin qui trovato e si continua il confronto finché si trova un nuovo minimo. Quando lo si trova, lo si scrive sul foglio di carta assumendolo come nuovo minimo e così via.

La domanda che dobbiamo porci[6] ora è la seguente: "Qual è stato il processo che ci ha permesso di trovare l'algoritmo?"

Naturalmente nel caso dell'autore la risposta è facile: sapeva già farlo. Bella forza. Ma forse, qualche studente inizialmente confuso, dopo un'iniziale suggerimento è giunto autonomamente a determinare un proprio algoritmo, più o meno simile a quello proposto. Quindi: "Qual è stato il processo che ha portato lo studente inizialmente confuso a determinare autonomamente l'algoritmo?".

Credo possa essere stato, per alcuni, rappresentare *prima* il problema in forma banale e *poi* in forma più complessa. Normalmente lo studente non segue questo processo, ovvero non si avvicina per gradi alla soluzione, ma la cerca "tutta insieme". Un simile approccio raramente dà frutti.

Un altro espediente utile è quello di usare "carta e penna". Inutile utilizzare subito strumenti formali (diagrammi di flusso, pseudocodifica, linguaggi evoluti, ecc.) di cui magari non si è perfettamente padroni. Meglio usare strumenti informali che ci permettono di scarabocchiare, prendere appunti, fare prove, calcolare espressioni e tutto ciò senza doversi concentrare sull'uso della carta e della penna di cui si è perfettamente padroni.

Sono convinto che troppi studenti si sottovalutino e si facciano scudo con il "genietto" della classe. Con ogni probabilità "quello bravo" non è un genio, ma spesso è solamente una persona abituata a *stabilire connessioni fra i suoi saperi*. Il termine "genio" andrebbe riservato per persone come Gauss o Mozart, che già in tenera età diedero incontrovertibili prove della loro intelligenza.

Non abbia quindi paura lo studente a cimentarsi con problemi che sfidano la sua intelligenza. Quando si affrontano detti problemi, si vince sempre, anche quando non si trova un risultato "utile". Si vince perché ci si mette in gioco e si esce dall'angolo. Questa è la più bella vittoria che uno studente possa sperimentare a scuola: provarci.

Ora, però, è bene tornare al problema iniziale, di cui si è discusso una possibile soluzione e la si è illustrata mediante il linguaggio naturale.

Come al solito, si deve provare a tracciare un diagramma di flusso. L'operazione non dovrebbe essere impossibile, dato che un'idea di quale possa essere la soluzione già c'è.

Un esempio di come possa svilupparsi il diagramma di flusso è indicato in fig. 8.5 a fronte. In tale diagramma appare evidente come la prima azione da eseguire sia proprio *assumere un riferimento iniziale*, ovvero stabilire che un determinato numero, ad esempio il primo, diventi il temporaneo minimo. Ciò deve avvenire fuori dal ciclo `for`, come è evidenziato graficamente.

[6]Non nel senso di maiali.

Eseguita detta azione si entra nella struttura del `for` inizializzando l'indice (è sempre la prima azione che viene eseguita nel `for`) e successivamente si testa la condizione booleana di uscita dal ciclo. Si noti che l'indice viene inizializzato a 1 e non a 0, perché il primo dei numeri da digitare è stato immesso *prima* di entrare nel ciclo.

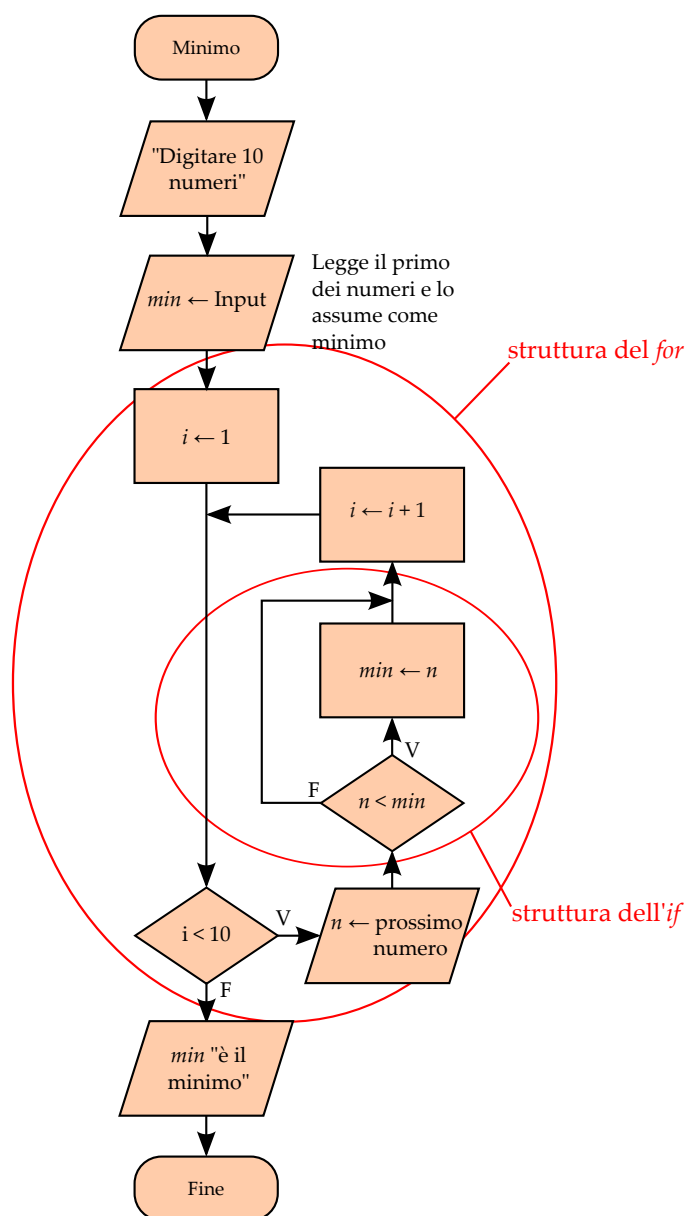



Figura 8.5: Diagramma di ricerca del minimo

Se l'espressione booleana è *non falsa* si legge il prossimo numero e lo si confronta con il minimo parziale all'interno di una struttura condizionale: se il nuovo numero è minore del minimo parziale si aggiorna il minimo parziale,

altrimenti non si esegue alcuna azione. Alla fine della struttura condizionale, ovvero al termine del corpo del ciclo, viene incrementato l'indice. Si noti che ciò avviene automaticamente, senza bisogno che venga posta un'apposita istruzione a fine corpo: viene esemplicemente eseguita la terza espressione del `for`, ogni qualvolta si completa il corpo del ciclo. Quando tutte le iterazioni del ciclo sono terminate, `min` conterrà il minore dei numeri digitati e si può mandare in *output* il risultato.

Se il diagramma è chiaro e non presenta punti oscuri si può passare al codice in linguaggio C, che è null'altro che una semplice traduzione dal diagramma al C. Anche in questo caso si presenta solamente il codice relativo all'algoritmo e non l'intero programma.

Listing 8.6: Ricerca del minimo

```
const int kNumeri = 10;                 //Numeri da digitare

//Prima parte dell'algoritmo:si visualizza la frase di cortesia
//e si digita il primo dei numeri prendendolo come riferimento.
printf("Digitare␣%d␣numeri\n", kNumeri); //Frase di cortesia
scanf("%d", &min);                      //Immette riferimento

//Struttura del ciclo for. Qui avviene il confronto ciclico dei
//numeri digitati con il riferimento. Si noti come l'indice sia
//inizializzato a 1 e non a 0. Cio' e' dovuto al fatto che il
//primo numero e' stato digitato fuori dal ciclo.
for (i=1; i<kNumeri; i++)
{
   //Digita il prossimo numero
   scanf("%d\n", &n);

   //Confronta il numero appena digitato con il minimo parziale
   //e se il numero digitato e' minore, aggiorna detto minimo.
   if(n<min)
      min = n;  //Aggiornamento del minimo parziale
}

//Stampa il minore dei numero digitati
printf("Il␣minore␣dei␣numeri␣digitati␣e'␣%d", min);
```

Si consiglia allo studente di studiare il presente esercizio, ossia a chiedersi il "perché" dei vari passaggi, sia dell'algoritmo, che del diagramma che del codice. Non ci si limiti a "dare una guardata" o leggere l'esercizio. Studiare significa chiedersi "perché" e tentare di darsi delle risposte.

Infine, prima di concludere l'esercizio si vuole richiamare ancora una volta l'attenzione dello studente sui commenti presenti nel programma. Essi rappresentano la parte quantitativamente forse più importante. Non si tratta di una deformazione scolastica della realtà, ma un modo professionale di programmare. Ci si rende conto che il Web è pieno di programmi non commentati, ma questo rappresenta soltanto un ulteriore motivo per soppesare criticamente la qualità presentata in taluni siti Web. Il commento è essenziale e insostituibile. Un programma non commentato nasce morto e dal punto di vista professionale non sarà possibile né effettuare manutenzione né eseguire un qualsiasi *upgrade* futuro, se non si attuano sforzi sproporzionati per l'intervento.

8.4 L'iterazione a scelta finale (ciclo *do..while*)

L'iterazione a numero di cicli non definito a priori e a scelta finale è sicuramente una struttura sintatticamente più semplice rispetto al ciclo `for`. Il concetto del ciclo `do..while` è la ripetizione di una sequenza che viene eseguita almeno una volta e che termina all'avverarsi di una condizione booleana. Affinché il ciclo possa terminare è fondamentale che nell'espressione booleana vi sia almeno una variabile in grado di determinare la condizione di uscita e che questa venga modificata nel corpo del ciclo[7].

Una possibile rappresentazione grafica del ciclo *do..while* potrebbe essere quella indicata in fig. 8.6.

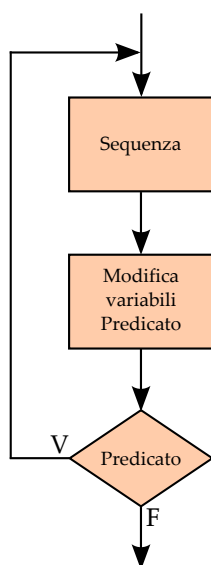



Figura 8.6: Iterazione con numero di cicli non definito e scelta finale

Dal diagramma appare evidente che si entra nel ciclo *sempre almeno una volta* e che si rientra nel ciclo se la condizione booleana è *non falsa*. Si ribadisce la necessità di modificare nel corpo del ciclo almeno una variabile dell'espressione booleana, altrimenti non si realizzerà mai la condizione di uscita. Questo concetto verrà presto illustrato attraverso un esempio.

Nel diagramma il corpo del ciclo evidenzia separatamente la sequenza e la modifica del valore del predicato. E' bene aggiungere qualche parola a tal proposito.

La sequenza, in realtà, potrebbe contenere qualsiasi altro costrutto: selezioni, cicli, strutture miste o addirittura, in certi casi, vuote. Essa rappresenta semplicemente il corpo del ciclo, senza voler indicare con precisione una determinata struttura. Inoltre, nel diagramma il blocco di modifica del predicato è evidenziato separatamente dalla sequenza. Si tratta di una semplificazione: semplicemente si vuole sottolineare la necessità, a meno che non si voglia scrivere un ciclo infinito (si veda il terzo esempio del codice presentato in 8.2 a pagina 224), di modificare le variabili che influenzano il valore del predicato.

[7]Si suppongono assenti eventi asincroni capaci di modificare dette variabili.

Dal punto di vista sintattico, invece, l'iterazione a numero di cicli definito e a test finale ha la seguente struttura:

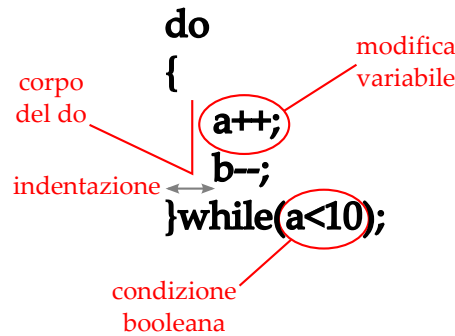



Figura 8.7: Struttura del ciclo *do..while*

In figura è presentato un esempio di modifica di variabile del predicato. Si supponga, per semplicità, che il valore della variabile `a` sia inferiore a 10. In tal caso l'espressione booleana sarà vera e si rientrerà nel corpo del ciclo. Modificando il valore della variabile che condiziona il predicato, e più precisamente incrementandola, prima o poi essa assumerà il valore 10, rendendo falsa l'espressione booleana e permettendo l'uscita dal ciclo.

Si noti che se la variabile `a` venisse decrementata, anziché incrementata, (concettualmente) non si uscirebbe dal ciclo[8].

L'istruzione di incremento della variabile `a` è quindi indispensabile per terminare il ciclo: se ci fosse solamente il decremento della variabile `b` nel corpo del `do`, la condizione bolleana non diventerebbe mai falsa e non si uscirebbe mai dal ciclo.

8.5 Esercizi svolti con il ciclo *do..while*

Il ciclo *do..while*, in qualità di iterazione a numero di cicli non definito e test finale, si presta meravigliosamente bene per illustrare alcune tecniche elementari di programmazione. In particolare se ne illustrerà una nel prossimo esercizio.

Esercizio - ◊◊◊ Inserisci un numero

Si supponga di dover acquisire un numero da tastiera che sia minore di 10 ma maggiore di 3. Se il numero digitato non rispetta i requisiti indicati si deve reinserirlo, altrimenti lo si stampa.

Soluzione

Il presente esercizio è molto semplice, ma rappresenta, a suo modo, un classico. Ogni qualvolta si presenta la necessità di acquisire dei dati che abbiano determinati requisiti, può essere molto utile fare riferimento a questo esercizio.

[8]Si è aggiunto un "concettualmente" perché in realtà dal ciclo si esce. Infatti la variabile verrebbe decrementata fino a diventare negativa e poi ancora fino ad assumere il valore -2^{31}, ossia -2147483648 che in esadecimale equivale a 0x80000000. Decrementando ulteriormente tale valore di ottiene 0x7FFFFFFF che rappresenta in decimale il valore +2147483647, ovvero $+2^{31}-1$ che è il massimo valor positivo espresso in complemento a due su 32 bit.

Lo studente potrebbe chiedersi cosa c'entri il ciclo in un esercizio come questo. In effetti esso nasconde una ripetizione: l'utente, chiamato a digitare un numero maggiore di 3 e minore di 10, potrebbe digitare 5, nel qual caso il numero di "ripetizioni" si ferma a 1. Potrebbe, però, per errore, digitare 2 la prima volta e 7 la seconda, nel qual caso le ripetizioni sarebbero 2. L'utente potrebbe essere uno studente straniero appenna arrivato in Italia e ancora in difficoltà con la lingua, per cui le ripetizioni potrebbero salire a 3-4 e così via.

Quindi sicuramente si avrà la necessità di utilizzare un ciclo. Di conseguenza dobbiamo chiederci *se è noto il numero delle ripetizioni*, ovvero il numero delle volte che si digiterà il numero. Naturalmente non è possibile saperlo: con ogni probabilità, nella stragrande maggioranza dei casi il numero delle ripetizioni sarà 1, ma potrebbero essere di più a seconda del livello di distrazione o anche semplicemente della bontà del collaudo che si intende applicare. Siccome non si conosce il numero delle ripetizioni dobbiamo escludere il ciclo `for`.

Dobbiamo allora chiederci se esiste la possibilità di *non entrare mai nel ciclo*, ovvero se il numero delle ripetizioni può essere zero. Anche questa eventualità è da escludere, dato che *almeno una volta il numero va digitato*. Ciò ci porta a concludere che, dovendo entrare almeno una volta nel corpo del ciclo, dovremo utilizzare un ciclo a test finale, ossia un `do..while`.

Si vuole sottolineare con forza che un qualsiasi altro ciclo *sarebbe concettualmente errato*. Si noti che, secondo Böhm e Jacopini, sarebbe utilizzabile anche un ciclo `while`, ma sarebbe sbagliata la scelta. Il linguaggio C utilizza tre tipi diversi di cicli per coprire tre problematiche concettuali diverse, ed è corretto correlare ciascuna di tali casistiche al rispettivo ciclo.

Alla luce di quanto detto, un possibile diagramma di flusso potrebbe essere il seguente:

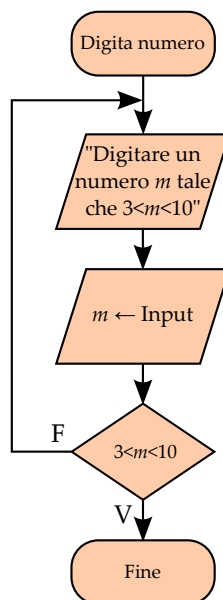



Figura 8.8: Diagramma Digita numero

Dal diagramma appare evidente la necessità di entrare almeno una vol-

ta nel corpo del ciclo prima di valutarne l'eventuale uscita mediante il test dell'espressione booleana.

Inoltre, appare anche evidente come il corpo del ciclo contenga anche l'*input* della variabile m che fa parte dell'espressione booleana di valutazione dell'uscita dal ciclo.

Tutte le caratteristiche che un buon ciclo `do..while` deve possedere sono soddisfatte, per cui si può passare ad analizzare il codice.

Listing 8.7: Digita numero

```
do
{
   //Frase di cortesia
   printf("Digita un numero tale che esso\n");
   printf("sia maggiore di 3 e minore di 10: ");

   //Immissione del numero intero m. Si noti
   //la sintassi dell'espressione booleana del
   //sottostante while.
   scanf("%d", &m);
}while((m>3)&&(m<10));
```

In caso di errore viene sempre visualizzata la stessa monotona frase. Naturalmente il codice è modificabile secondo la propria sensibilità. Vuole solo essere un esempio. L'importante è che sia presente la struttura `do..while`.

Molti studenti commettono l'errore di omettere il ciclo, quando si tratta di immettere numeri che debbano possedere determinati requisiti. Ciò è un palese errore e per giunta piuttosto diffuso e grave. E' bene evitarlo.

Il prossimo esercizio è di relativa lunghezza e complessità. Esso avrà la forma di un programma completo e richiede una certa attenzione da parte dello studente.

Esercizio - ◊◊♦ Numero di zeri

Si chiede di digitare un numero naturale qualsiasi e di stampare a video il numero di zeri significativi di cui è composto. Ad esempio, il numero 010590480 possiede i requisiti richiesti e contiene 3 zeri significativi.

Soluzione

Solitamente il presente esercizio crea qualche problema allo studente non esperto. E' bene, quindi, esaminare bene il testo prima di incamminarsi su una strada errata e solo poi formulare qualche ipotesi di soluzione.

La prima richiesta che il testo formula è la digitazione su tastiera di un numero naturale qualsiasi. Ciò significa un numero $n \in \mathbb{N}$. Si dovrà quindi porre cura affinché il numero intero digitato sia non negativo. La seconda parte del testo non pare nascondere insidie, anche se potrebbe risultare di più difficile soluzione rispetto alla prima parte. Si dovrà valutare cifra per cifra il numero digitato ed incrementare un contatore ogni qualvolta si rileva la cifra zero.

Se il testo è stato correttamente interpretato e non ci sono dubbi residui, si potrebbe tentare, alla luce di quanto detto, di tracciare un primo diagramma di

flusso, molto grezzo, in modo da suddividere il problema complessivo in sottoproblemi più semplici da affrontare. Il diagramma potrebbe essere formato dai seguenti blocchi:

1. l'immissione del numero naturale;
2. il conteggio degli zeri;
3. la stampa del risultato.

Graficamente ciò si traduce nel seguente diagramma di flusso:

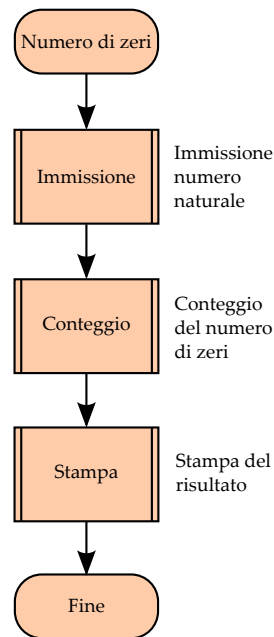



Figura 8.9: Numero di zeri: Diagramma sommario

Si noti, però, che il blocco di **Stampa** è veramente banale, per cui probabilmente conviene porre la stampa del risultato direttamente nel blocco **Conteggio**, anche se da un punto di vista logico la suddivisione in tre parti è corretta.

Si apre una brevissima parentesi. La notazione grafica utilizzata per i tre blocchi di elaborazione è quella relativa alle *subroutines* che corrispondono alle funzioni nel linguaggio C. I puristi potrebbero storcere il naso davanti ad un uso improprio della simbologia. Non si vuole indicare tecnicamente una funzione con il simbolo usato, ma semplicemente una porzione di programma.

Il primo blocco di elaborazione, denominato **Immissione**, non sembra complicato. Esso raggruppa le azioni che permettono all'utente di digitare un numero naturale qualsiasi. Si dovrà, come già detto, prestare attenzione affinché il numero sia non negativo, il che implica l'uso di un ciclo, dato che ci si potrebbe sbagliare a digitare il numero.[9]

Si deve quindi determinare di quale ciclo si tratti: siccome il problema è del tutto simile a quello dell'esercizio precedente si opta per un ciclo `do..while`.

[9] In realtà la probabilità che l'utente si sbagli è remota. Quello che si vuole creare è un cosidetto programma a "prova d'idiota", che prevede che l'utente possa sbagliare.

Anche il diagramma di flusso è del tutto simile a quello presentato per l'esercizio precedente: cambia leggermente la condizione booleana da verificare.

Fatte le debite premesse, il diagramma di flusso del primo blocco potrebbe assumere una struttura simile a quella di fig. 8.10.

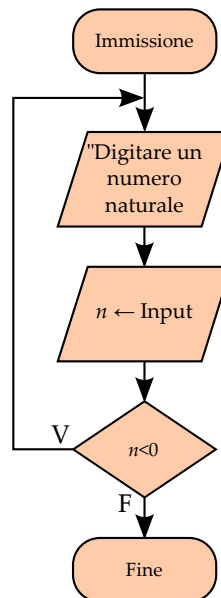



Figura 8.10: Numero di zeri: Immissione

Siccome si immagina che il presente diagramma non generi problemi di alcun tipo, si potrebbe scrivere il codice relativo. Anche in questo caso si apre una breve parentesi. Si dovrebbero scrivere piccole porzioni di codice e *debuggarle*, soprattutto se si tratta di codice complessivamente voluminoso. Per questioni grafiche nelle presenti pagine si presentano solo gli spezzoni di codice relativi ai vari blocchi, salvo poi, alla fine, proporre il programma completo di tutte le sue parti. Lo studente farebbe bene però a scrivere un primo programma contenente solo la prima parte e testarlo. Quando funziona aggiungere una seconda parte e così via. Per i motivi suddetti si procederà in maniera diversa.

Il codice relativo al diagramma di fig. 8.10 è proposto di seguito:

Listing 8.8: Immissione

```
do
{
   //Si spiega brevemente cosa fa il programma
   printf("Il␣presente␣programma␣conta␣di␣quanti␣zeri\n");
   printf("e'␣composto␣un␣numero␣naturale␣digitato.\n\n");

   //Invita l'utente a digitare il numero naturale
   printf("Digita␣un␣numero␣naturale:␣");

   //Immissione del numero naturale n.
   scanf("%lld", &n);   //Si noti la sintassi
}while(n<0));
```

La definizione della variabile `n` non è esplicitata, ma la si può supporere di tipo `long long`, al fine di contenere il maggior numero di cifre possibili.

La seconda parte del programma è sicuramente più complicata, anche e soprattutto dal punto di vista dell'algoritmo. Anche in questo caso si coglie l'occasione per commentare l'approccio metodologico. Durante lo svolgimento dell'esercizio "**Trova il minimo**" proposto a pag. 228 si è consigliato di semplificare l'esercizio per poi aumentare la difficoltà e generalizzare il problema. Incontriamo però sempre il solito problema: abbiamo un cervello troppo potente. Si vedano i due numeri proposti di seguito:

10
208160575340925665004487

Non abbiamo nessuna difficoltà a contare nè il numero di zeri del primo numero, formato di sole 2 cifre, né quelli del secondo numero formato da ben 24 cifre: il primo numero contiene uno zero ed il secondo numero contiene cinque zeri. E aumentando il numero di cifre del secondo numero non aumenta la difficoltà che il nostri cervello incontra ad isolare le singole cifre.

Quindi il metodo proposto nel predetto esercizio stavolta non funziona. Finché non riusciamo a mettere in difficoltà il nostro cervello ci risulta difficile anche scomporre le azioni che eseguiamo. Semplicemente "vediamo" le singole cifre senza bisogno di fare calcoli o applicare algoritmi. Quindi che strategia utilizzare?

Uno stratagemma che in classe ha portato spesso frutti, è porre uno studente con le spalle alla lavagna e, dopo aver scritto il numero sulla medesima, esortare il malcapitato a formulare domande (che non contengano la parola "cifra") ai suoi compagni di classe, che naturalmente il numero lo possono vedere.

Ben presto lo studente inizierà a formulare domande relative alle quattro operazioni aritmetiche e da quel momento in poi arrivare alla soluzione è solo questione di minuti.

Si lascia del tempo allo studente che voglia cimentarsi sfruttando l'idea. Nel frattempo si può aggiungere un commento. Dando per risolto il problema, lo studente potrebbe rendersi conto che non vi sono stati grandi cambi di strategia: si è messo in difficoltà il cervello impedendogli di "vedere" le cifre. Ciò ci obbliga ad astrarre il problema e non usare "mezzi illeciti" quali la visione. Dal punto di vista visivo è naturale isolare dal contesto i singoli elementi di un'immagine, quindi anche le cifre di un numero. Tale azione non ha, però, nulla in comune con ciò che riusciamo ad eseguire con il microprocessore avendo per *input* una tastiera e per *output* un video.

Con queste premesse lo studente dovrebbe abituarsi ad usare solamente strumenti elementari per tentare di risolvere dei problemi utilizzando il linguaggio C, ovvero gli stessi strumenti forniti dal linguaggio: gli operatori aritmetico-logici, le tre strutture fondamentali, ecc.

Nel caso specifico possono esserci utili proprio gli operatori aritmetici, dato che vi sono sostanzialmente due modi per "estrarre" le singole cifre da un numero intero.

Si supponga, per semplicità il numero naturale 1234 e di volerne estrarre le singole cifre. Un metodo banale potrebbe essere il seguente:

1. dividere il numero n per 10, mediante divisione intera;
2. moltiplicare il numero così ottenuto m per 10, ottenendo p;
3. sottrarre p da n: il risultato c sarà la cifra meno significativa di n;
4. se m è uguale a zero l'algoritmo termina, altrimenti...
5. ...assegnare a n il valore di m e tornare al punto 1.

Detto così sembra piuttosto complicato, ma facendo un esempio concreto dovrebbe diventare tutto più chiaro. A tal fine è bene ricordare una massima di D.E.Knuth che pone in evidenza un frequente errore dello studente medio:

> *"Gli algoritmi vanno eseguiti, non studiati"*
>
> Donal E. Knuth

Quindi, fidandoci ciecamente di uno dei maggiori divulgatori informatici esistenti, si procede eseguendo l'algoritmo appena formulato:

Primo algoritmo

1. *dividere il numero n per 10, mediante divisione intera*

$$m \leftarrow n/10 \tag{8.3}$$

$$1234/10 = 123 \tag{8.4}$$

2. *moltiplicare il numero così ottenuto m per 10, ottenendo p*

$$p \leftarrow m \cdot 10 \tag{8.5}$$

$$123 \cdot 10 = 1230 \tag{8.6}$$

3. *sottrarre p da n: il risultato r sarà la cifra meno significativa di n*

$$r \leftarrow n - p \tag{8.7}$$

$$1234 - 1230 = 4 \tag{8.8}$$

4 - 5. *se m è uguale a zero l'algoritmo termina, altrimenti assegnare a n il valore di m e tornare al punto 1*

$$n \leftarrow m \tag{8.9}$$

e siccome $m \neq 0$ si torna al punto 1.

L'algoritmo presentato funziona perfettamente utilizzando solamente la divisione intera, la moltiplicazione e la sottrazione. Esiste, però, un secondo modo, più efficiente per ottenere gli stessi risultati. E' infatti possibile calcolare direttamente il resto di una divisione intera mediante l'operatore `%`.

L'uso di detto operatore rende l'algoritmo più efficiente, come si cercherà di evidenziare nel secondo esempio:

Secondo algoritmo

1. *calcolare il resto r della divisione intera* $n/10$ *contenente il LSD* n;
2. *dividere n per 10;*
3. *se n è uguale a 0, l'algoritmo termina...*
4. *...altrimenti si torna al punto 1.*

Siccome il secondo algoritmo è più efficiente del primo si adotterrà quest'ultimo. Naturalmente, alla fine del punto 1 si dovrà valutare se il resto r appena calcolato, che rappresenta la cifra meno significativa del numero (che ad ogni iterazione "perde" una cifra), vale zero oppure no.

Ogni qualvolta il resto r è uguale a zero, si dovrà incrementare un contatore che rappresenterà il numero degli zeri significativi di cui è composto il numero originale.

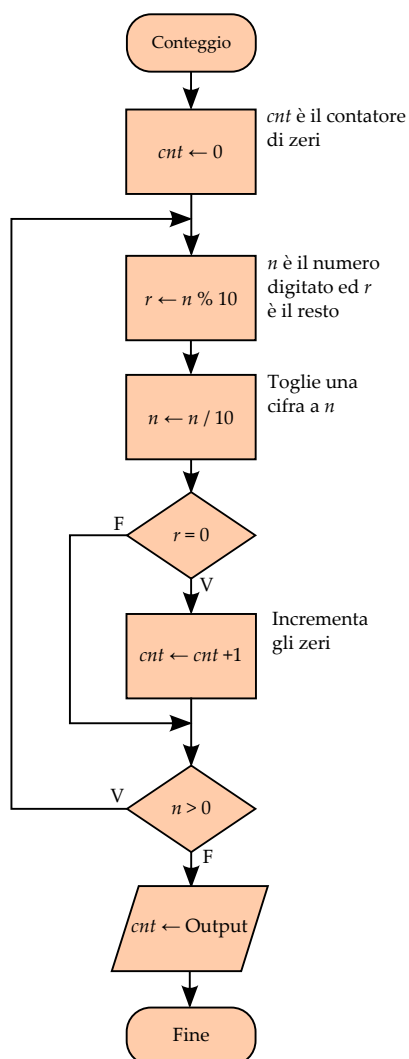



Figura 8.11: Numero di zeri: Conteggio

In entrambi i casi, comunque si può intravvedere un ciclo a test finale, come suggerisce l'ultimo punto di ciascun algoritmo. Alla luce di ciò è possibile tracciare il diagramma di flusso di detta parte del problema, rappresentato in fig. 8.11.

Come si nota è stato aggiunto il blocco di stampa del risultato al termine dell'algoritmo, data la intrinseca banalità di detta azione. A parte cio, il dia-

gramma di flusso rispecchia il secondo algoritmo illustrato mediante linguaggio naturale e si ritiene che non dovrebbe presentare eccessivi problemi.

E' quindi possibile passare dal diagramma al codice C. Anche in questo caso si presenta il solo codice relativo al diagramma, ricordando che `n` è di tipo `long long`, mentre le variabili `r` e `cnt` possono essere benissimo di tipo `char`, dato che la prima non assumerà mai valori superiori a 9 e dato che una variabile di tipo `long long` non potrà mai contenere più di 128 zeri significativi, dato che un numero intero di tipo `long long` può essere formato al massimo da 18 cifre.

Il codice assume la seguente struttura:

Listing 8.9: Conteggio

```
cnt = 0;         //Azzera il contatore di zeri
t = n;           //Salva n perche' verra' distrutta

//Ciclo di "estrazione" della singola cifra
//e di conteggio degli zeri
do
{
   //Estrae la singola cifra e la pone in r
   r = n%10;

   //Diminuisce di una cifra n
   n /=10;

   //Se r = 0, incrementa il contatore di zeri
   if(r==0)
      cnt++;     //La cifra era uno zero
}while(n>0);     //Continua finche' ci sono
                 //cifre significative in n

//La stampa e' divisa in due parti per motivi
//grafici. Si noti la sintassi delle specifiche
//di conversione.
printf("Il␣numero␣%lld␣contiene␣", t);
printf("%hhd␣zeri␣significativi", cnt);
```

L'unica differenza degna di nota con il diagramma è data dalla variabile temporanea `t`, atta a salvare il valore originale di `n`. Ciò si rende necessario in fase di stampa del risultato.

A questo punto è possibile raccogliere l'intero codice in un programma completo:

Listing 8.10: Numero di zeri (Completo)

```
void main()
{
   long long int n;     //Contiene il numero da valutare
   long long int t;     //Variabile temporanea
   char r;              //Contiene la singola cifra estratta
   char cnt = 0;        //Contiene il numero di zeri

   //Immissione del numero di cui si deve valutare il
```

```
   //numero di zeri significativi. Siccome potrebbe
   //essere immesso un numero minore di zero, si usa
   //un ciclo.
   do
   {
      //Si spiega brevemente cosa fa il programma
      printf("Il␣presente␣programma␣conta␣di␣quanti␣zeri\n");
      printf("e'␣composto␣un␣numero␣naturale␣digitato.\n\n");

      //Invita l'utente a digitare il numero naturale
      printf("Digita␣un␣numero␣naturale:␣");

      //Immissione del numero naturale n.
      scanf("%lld", &n);
   }while(n<0);

   //Si estraggono le singole cifre del numero e si valuta
   //se sono uguali a zero.
   t = n;              //Salva n perche' verra' distrutta

   //Ciclo di "estrazione" della singola cifra
   //e di conteggio degli zeri
   do
   {
      //Estrae la singola cifra e la pone in r
      r = n%10;

      //Diminuisce di una cifra n
      n /=10;

      //Se r = 0, incrementa il contatore di zeri
      if(r==0)
         cnt++; //La cifra era uno zero
   }while(n>0); //Continua finche' ci sono
                //cifre significative in n
   //La stampa e' divisa in due parti per motivi
   //grafici.
   printf("Il␣numero␣%lld␣contiene␣", t);
   printf("%hhd␣zeri␣significativi", cnt);
}
```

Il programma è null'altro che la riunione dei singoli due spezzoni già visti, per cui non si ritengono necessari ulteriori commenti.

8.6 L'iterazione a scelta iniziale (ciclo *while*)

L'iterazione a numero di cicli non definito a priori e a scelta iniziale è il più flessibile dei tre cicli. Non a caso era *il* ciclo indicato da Böhm e Jacopini.

Il concetto del ciclo `while` è la ripetizione di una sequenza che *potrebbe non essere eseguita mai*. Come nel ciclo `do..while`, affinché il ciclo possa terminare è fondamentale che nell'espressione booleana vi sia almeno una variabile in grado di determinare la condizione di uscita e che questa venga modificata nel corpo del ciclo.

Una possibile rappresentazione grafica del ciclo *while* potrebbe essere quella indicata in fig. 8.12.

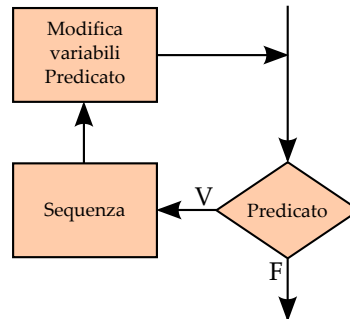



Figura 8.12: Iterazione con numero di cicli non definito e Scelta iniziale

Anche in questo caso si sono tenuti separati il blocco relativo alla sequenza e quello relativo alla modifica del valore del predicato. Tale scelta ha il solo scopo di porre in evidenza la necessità di modificare il valore del predicato nel corpo del ciclo.

La sintassi del ciclo `while` è rappresentata in fig. 8.13:

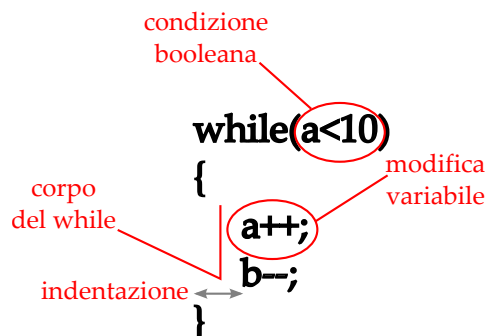



Figura 8.13: Struttura del ciclo while

Naturalmente si suppone che la variabile `a` sia stata adeguatamente inizializzata *prima* del ciclo e che il test iniziale si possa considerare assolutamente valido e lecito.

Proprio il test iniziale rende questo ciclo particolarmente flessibile ed adattabile. Mediante esso è possibile simulare validamente un ciclo `for` oppure un ciclo `do..while`. Naturalmente bisogna sempre chiedersi perché “simulare” qualcosa quando si può usare l’originale, ossia il ciclo `for` o il ciclo `while`.

8.7 Esercizi svolti con il ciclo *while*

Anche il ciclo *while*, in qualità di iterazione a numero di cicli non definito e test iniziale, si presta bene per illustrare alcune tecniche elementari di programmazione. Si vedano a tal proposito i prossimi esercizi.

Esercizio - ◊◊◊ Numero di cifre

Dato un numero intero non negativo si vuole conoscere il numero di cifre di cui è formato. Si tracci il diagramma di flusso dell'algoritmo e si scriva il relativo codice in linguaggio C.

Soluzione

Se il numero intero non negativo n è minore di 10, sicuramente avrà una sola cifra, altrimenti basterà dividere detto numero per 10 finché si ricade nella condizione iniziale, ovvero $n < 10$. Le divisioni ripetute indicano un'iterazione. Non sapendo a priori quante saranno dette divisioni, l'iterazione dovrà essere a numero di cicli non definito. Siccome le divisioni potrebbero essere anche 0, l'iterazione a numeri di cicli non defintо sarà a scelta iniziale.

Un possibile diagramma di flusso potrebbe essere il seguente:

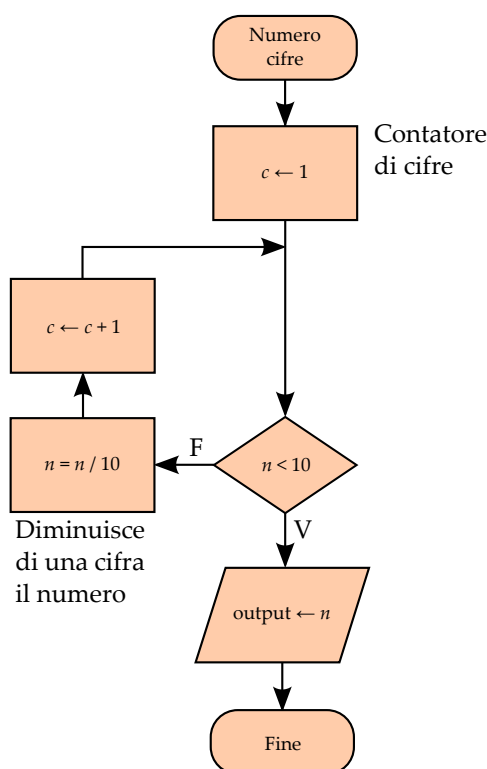



Figura 8.14: Numero cifre

Prima di entrare nel ciclo si inizializza il contatore di cifre a 1, dato che qualsiasi numero deve avere almeno una cifra. Dopodiché si entra nel ciclo a test iniziale, testando proprio se il numero di cifre è 1. In caso affermativo si esce dal ciclo (nel quale non si era mai entrati) e si stampa il risultato. Altrimenti si divide il numero per 10, si incrementa il numero di cifre e si riesegue il test.

Si noti che nel diagramma di flusso si (ri)entra nel corpo del ciclo se la condizione booleana è *falsa*, mentre nel ciclo `while` di entra nel corpo del ciclo quando la condizione booleana è *vera*. Non c'è contraddizione in ciò, dato

che il diagramma di flusso è indipendente dal linguaggio nel quale viene poi tradotto. Un possibile codice in linguaggio C potrebbe essere il seguente:

Listing 8.11: Numero cifre

```
c = 1;          //Inizializza numero di cifre
while(n>9)
{
   n /= 10;     //Diminuisce di una cifra il numero
   c++;         //Incrementa contatore di cifre
}

//Stampa il risultato
printf("Il numero %d e' composto ", n);
printf("da %d cifre", c);
```

Si noti che si è scelto di risolvere il problema dato mediante un ciclo `while` impostando il contatore di cifre a 1. In maniera altrettanto logica si sarebbe potuto scegliere un ciclo `do..while` impostando il contatore di cifre a 0.

Esercizio - ◊◊◊ Congettura di Collatz

Sia dato il seguente algoritmo:

1. sia dato un numero intero positivo n;
2. se n è dispari, si assegni ad n il risultato del calcolo $3n + 1$;
3. se n è pari si assegni ad n il risultato del calcolo $n/2$;
4. se $n = 1$ l'algoritmo termina, altrimenti si torna al punto 2.

Secondo il matematico tedesco Lothar Collatz, il suddetto algoritmo giunge sempre a termine. Ciò, però, non è stato ancora dimostrato.

Soluzione

L'ultimo punto dell'algoritmo suggerisce, se n è diverso da 1, di tornare al punto 2. Ciò indica inequivocabilmente la presenza di un ciclo (vedi fig. 8.15).

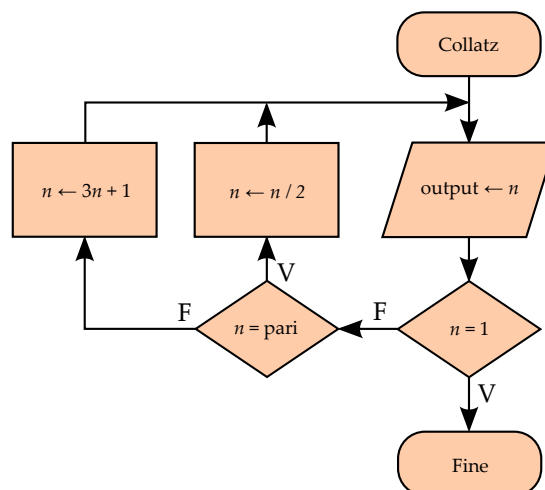



Figura 8.15: Collatz

Siccome non è possibile determinare il numero di cicli, si dovrà utilizzare un'iterazione con numero di cicli non definito a priori. Infine, siccome il numero positivo n potrebbe assumere inizialmente il valore 1, si devrà utilizzare un'iterazione a test iniziale.

Si fa notare, però, che il diagramma di fig. 8.15 a fronte non è esattamente rispondente a quanto indicato. Non è un ciclo a test iniziale, tanto per incominciare. Si potrebbe ipotizzare che sia un ciclo a test finale. Anche in questo caso, però, il diagramma non è perfettamente rispondente a un'iterazione a test finale, dato che la selezione testante la parità del numero dovrebbe comparire *prima* del test di uscita e non *dopo*.

Insomma il diagramma di fig. 8.15 non rispecchia esattamente un ciclo strutturato, nè a test finale nè a test iniziale. La cosa non deve stupire: un algoritmo tracciato mediante le tre strutture base non è detto che sia compatibile con i paradigmi della programmazione strutturata.

Un esempio di quanto detto è il diagramma di fig. 8.16a.

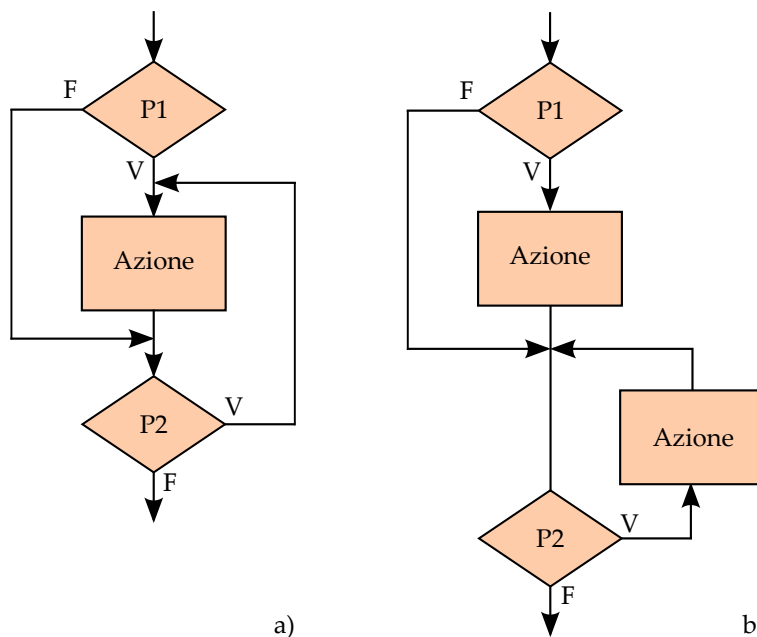



Figura 8.16: Struttura da evitare ... e corretta

Simili strutture non sono di per sé errate o da temere come la peste. Sono semplicemente da evitare per il semplice motivo che non esistono strutture equivalenti nei linguaggi ad alto livello, per cui ci si ritrova, all'atto della scrittura del codice, di non poter replicare in linguaggio C la struttura tracciata mediante il diagramma di flusso.

Il diagramma corretto e coerente con le strutture del linguaggio C (o di altri linguaggi evoluti) è indicato in fig. 8.16b, ove si riconosce una selezione basata sul predicato $P1$ seguita da un'iterazione a numero di cicli non definito e test iniziale, basata a sua volta sul predicato $P2$.

Un discorso del tutto analogo si può fare per il diagramma di flusso di fig. 8.15 a pagina 246. Un esempio di diagramma coerente con le strutture C potrebbe essere il seguente:

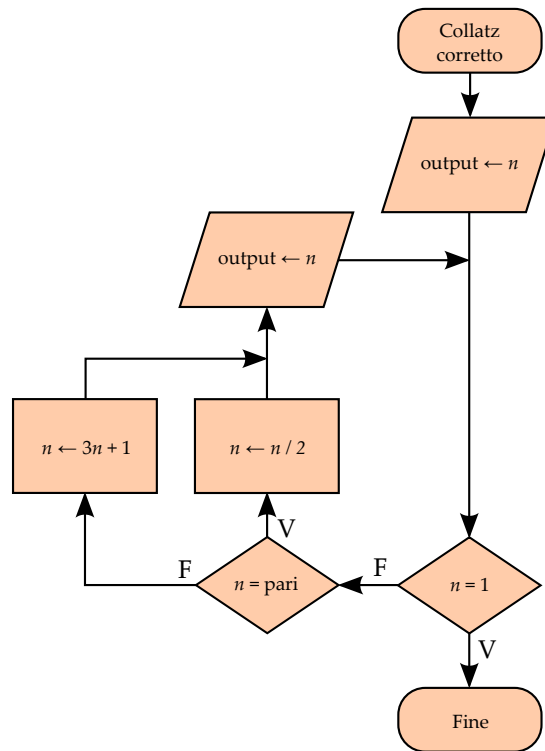



Figura 8.17: Collatz corretto

Il blocco di *output* è stato sdoppiato e un blocco è stato portato fuori dal ciclo, in modo da stampare *una tantum* il numero di partenza, mentre il secondo è stato portato *alla fine* del corpo del ciclo. In tal modo si è perfettamente coerenti con le strutture dei linguaggi ad alto livello e la traduzione in linguaggio C diventa banale. Si propone un esempio nel codice seguente:

Listing 8.12: Congettura di Collatz

```
printf("%d\n", n);//Si stampa solamente il numero,
                  //senza commenti
//Ciclo di esecuzione dell'algoritmo di Collatz.Il
//ciclo termina quando n=1.
while(n!=1)
{
   //Verifica se n e' pari, aggiorna n e stampa il
   //nuovo numero.
   if(n%2==0)
      n /= 2;
   else
      n = 3*n+1;
   printf("%d\n", n);
}
```

Non si ritiene necessario aggiungere ulteriori commenti al codice.

8.8 Un errore subdolo

Si vuole, nella presente sezione, trattare di un errore piuttosto frequente e comune al ciclo `while` e al ciclo `for`. Si veda il precedente codice, qui riprodotto insieme all'errore:

Listing 8.13: Congettura di Collatz con errore

```
printf("%d\n", n);//Si stampa solamente il numero,
                  //senza commenti
//Ciclo di esecuzione dell'algoritmo di Collatz.Il
//ciclo termina quando n=1.
while(n!=1);
{
   //Verifica se n e' pari, aggiorna n e stampa il
   //nuovo numero.
   if(n%2==0)
      n /= 2;
   else
      n = 3*n+1;
   printf("%d\n", n);
}
```

L'errore è dato dal punto e virgola dopo la condizione booleana del `while`. Detto punto e virgola *separa* di fatto lo *statement* `while` dal relativo corpo. Il risultato è un ciclo infinito dato che la variabile `n` non può essere mai modificata, non essendoci più alcun corpo del `while`.

Analogo discorso si può fare per il ciclo `for`. Di seguito è riprodotto il ciclo `for` relativo al primo esercizio della sezione 8.3:

Listing 8.14: Calcolo della serie con ciclo *for* ed errore

```
for (i=0, t=0.0, p=1; i<20; i++);
{
   t += 1.0/p;  //Si noti il numeratore
   p *= 2;      //Aggiorna il denominatore
}
```

Anche in questo caso il punto e virgola separa lo *statement* `for` dal corpo del ciclo. Il risultato è che l'istruzione `for (i=0, t=0.0, p=1; i<20; i++);` viene inutilmente eseguita 20 volte ed il corpo del ciclo una sola volta.

Si ponga estrema attenzione a simili (frequenti) errori: sfuggono facilmente alla vista del programmatore che legge ciò che vorrebbe essere rappresentato dal codice e non ciò che effettivamente rappresenta.

8.9 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 8.

Parte III

LA PROGRAMMAZIONE STRUTTURATA

Capitolo 9

I vettori

A volte è necessario trattare un consistente insieme di dati aventi tutti la stessa natura: le temperature rilevate nel corso di una giornata o di un mese; i caratteri che formano una frase; i termini di una successione aritmetica, ecc.

Tutti i dati elencati hanno due caratteristiche in comune:

1. sono in quantità finita;
2. sono tutti dello stesso tipo.

La prima delle due caratteristiche va commentata ulteriormente: in molti casi la quantità non è solo finita, ma è anche grossolanamente calcolabile il valor massimo. In presenza di un insieme di dati aventi le caratteristiche appena menzionate è possibile utilizzare una struttura dati particolare: il vettore.

Definizione 12 (Vettore)**.**
Il vettore è una struttura dati contenente un numero finito di variabili tutte dello stesso tipo.

Una rappresentazione grafica molto usata per descrivere il vettore è la seguente:

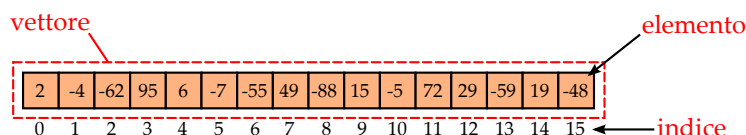



Figura 9.1: Il vettore

In figura 9.1 è rappresentato un vettore di interi (ad esempio di tipo `int`), formato da 16 *elementi* numerati da 0 a 15. Ogni elementi è leggibile/scrivibile mediante l'uso di un *indice* che indica l'elemento coinvolto nell'azione di lettura/scrittura. Ad esempio, il quarto elemento del vettore (ossia avente indice 3, dato che il primo elemento ha indice 0) assume il valore intero 95. Si sottolinea che il primo elemento di un qualsiasi vettore *ha sempre indice 0*.

9.1 Definizione di un vettore

Si supponga di voler definire un vettore di interi di 16 elementi, indicandolo con l'identificatore `vet`. La sintassi diventerebbe la seguente:

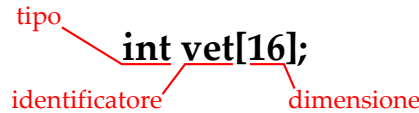



Figura 9.2: Definizione di vettore

Con riferimento alla fig. 9.2 il vettore è definito attraverso:

- un **tipo**;
- un **identificatore**;
- una **dimensione**.

Il tipo determina la natura degli elementi del vettore. Si sottolinea che specificando un solo tipo per un vettore dato *tutti* gli elementi del vettore apparterranno allo stesso tipo. L'indentificatore rappresenta il riferimento del vettore. Ogni qualvolta si deve operare sul vettore si utilizzerà il suo identificatore per accedere ad esso. Infine, la dimensione indica il numero di elementi del vettore. Tale dimensione deve essere indicata attraverso una costante intera.

Un errore frequentissimo fra gli studenti in fase di definizione di vettore consiste nell'indicare la dimensione dello stesso mediante una *variabile*. Ciò non è possibile! La dimensione di un vettore va indicata mediante una costante intera maggiore di 0 oppure mediante un'espressione di costanti intere avente valore risultante maggiore di 0. In nessun modo è possibile *definire* un vettore di dimensioni variabili. Ciò dipende dal fatto che l'allocazione dello spazio di memoria nello *stack* non avviene dinamicamente ma staticamente, per cui non può essere determinata *run time* ma deve essere determinabile dal preprocessore.

9.2 Lettura/scrittura di un elemento

Un errore terminologico molto frequente fra studenti suona più o meno così: "Leggendo il vettore...". Deve essere chiaro che non si può leggere o scrivere un vettore, ma solamente i suoi elementi presi singolarmente. Esiste una sola eccezione a quanto detto, che è la seguente forma di inizializzazione di un vettore:

Listing 9.1: Inizializzazione di vettore

```
int veta[] = {3, 7, -56, 49, 0, -4};
int vetb[6] = {3, 7, -56, 49, 0, -4};
```

Si noti, però, che le suddette forme sono applicabili *solo e solamente durante la definizione del vettore* e non a definizione o dichiarazione avvenuta. Si noti anche che nel primo caso non viene specificata la dimensione del vettore, che emerge *solo* dal numero di elementi posti fra parentesi graffe e separati da virgole,

mentre nel secondo caso la lunghezza del vettore viene esplicitamente indicata. Entrambe le notazioni sono corrette ed equivalenti. Nel secondo caso sarebbe stata corretta anche la notazione `int vetb[10] = {3,7,-56,49,0,-4};`. In tal caso sarebbero stati inizializzati solamente i primi 6 elementi del vettore.

La sintassi della lettura/scrittura mediante assegnazione di un elemento del vettore è la seguente:

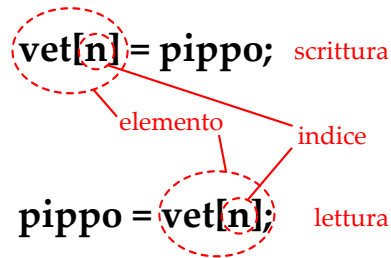



Figura 9.3: Lettura/scrittura di un elemento di un vettore

La prima condizione necessaria non sufficiente è che `pippo` sia dello stesso tipo del singolo elemento del vettore. La seconda condizione affinché la sintassi risulti corretta è che n sia di tipo intero. Si noti che l'indice che compare fra parentesi quadre può essere una costante, una variabile, una espressione o una funzione: l'importante è che il valore assunto sia di tipo intero.

E' importante sottolineare che *non viene fatto alcun controllo* sul valore di n, il che significa che si può, per errore, indicare un elemento che non esiste, leggendo o scrivendo fuori dal vettore. In entrambi i casi si è in presenza di un *bug* del codice.

9.3 Esempi d'uso dei vettori

I prossimi due esempi illustrano in modo pratico l'uso dei vettori e l'ambito di utilizzo tipico di detta struttura dati. Il primo esempio è un *must* di qualsiasi introduzione alla programmazione:

Esercizio - ◊◊◊ La successione di Fibonacci

Si chiede il diagramma di flusso e la scrittura del codice relativi all'algoritmo di produzione e memorizzazione in un vettore dei primi 100 elementi della successione di Fibonacci[1].

Quest'ultima è espressa come:

$$F_n = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F_{n-1} + F_{n-2}, & \text{se } n > 1. \end{cases} \tag{9.1}$$

[1]Il matematico toscano Leonardo Pisano, detto Fibonacci (*filius Bonacci*) visse a cavallo del XII e XIII secolo. Definì l'omonima successione nel trattato *Liber abaci* del 1202.

Soluzione

Volendo memorizzare i primi 100 termini della successione di Fibonacci senza utilizzare un vettore si dovrebbero definire ed utilizzare 100 variabili diverse, ciascuna delle quali con un proprio identificativo. Saremmo impossibilitati a usare un ciclo proprio per la necessità di cambiare 100 volte nome alla variabile da scrivere. Questo è un tipico esempio in cui il vettore è insostituibile: quando cioè si devono memorizzare in una struttura dati molti valori tutti dello stesso tipo.

Come al solito si chiede di affrontare il problema "cartesianamente", valutando, innanzi tutto, il testo e cercando di fugare ogni possibile dubbio riguardante le consegne.

La funzione 9.1 nella pagina precedente illustra che il primo termine della successione di Fibonacci vale 0 per definizione, e che analogamente il secondo termine vale 1. I termini successivi sono sempre dati dalla somma dei due precedenti. La successione, quindi, sembra crescere piuttosto lentamente, ma non è così. Quest'aspetto è estremamente importante per quanto riguarda la scelta del tipo di vettore. Qual è il valore che la successione raggiunge dopo 100 termini? La domanda è tutt'altro che secondaria, dato che il valore che assume il 100_{esimo} elemento della successione di Fibonacci non è rappresentabile nemmeno mendiante un `unsigned long long int`.

Lo studente curioso potrebbe a questo punto cercare sul Web una qualche espressione analitica per il calcolo dei termini della successione di Fibonacci ed imbattersi, senza dover fare ricerche approfondite, nella formula di Binet,[2] evidenziata in 9.2:

$$F_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{9.2}$$

Si tratta di una formula per certi versi sorprendente, dato che contiene dei termini irrazionali ma produce *sempre*, per qualsiasi valore di n, valori naturali. Utilizzando tale formula, apprendiamo che il valore che assume il 100_{esimo} termine è 218922995834555169026, mentre $2^{64} - 1 = 18446744073709551615$. Scopriamo anche che il 94_{esimo} termine vale 12200160415121876738, che è il massimo termine della successione rappresentabile mediante una variabile di tipo `unsigned long long int`. Quindi il testo del problema va rivisto, in modo da calcolare, ad esempio, solo i primi 90 termini della successione.

Si noti che lo studente avrebbe potuto giungere alle stesse conclusioni senza il bisogno della formula di Binet e senza dover calcolare a mano i primi 100 termini della successione di Fibonacci. In maniera molto più empirica si sarebbe potuto definire un vettore di tipo `unsigned long long int`, scrivere il programma che esegue il calcolo dei 100 termini della successione e analizzare gli ultimi elementi.

Quando un determinato termine non dovesse essere palesemente somma dei precedenti due significherebbe che detta somma avrebbe prodotto *overflow* e si sarebbe trovato il maggiore dei termini della successione di Fibonacci rappresentabile mediante una variabile di tipo `unsigned long long int`.

[2] Jacques Philippe Marie Binet, matematico e astronomo francese, la dimostrò nel 1843.

Il testo del problema è stato analizzato e si è cercato di fugare qualsiasi ambiguità o dubbio legato ad esso. Cartesio[3] propone a questo punto l'analisi del problema al fine di diminuirne la difficoltà d'approccio scomponendolo in sotto-problemi più facili da affrontare singolarmente.

Siccome dal terzo termine in poi ciascun valore è calcolabile mediante somma dei due precedenti, appare abbastanza chiaro che si è in presenza di un'iterazione, per cui la prima domanda alla quale si deve rispondere è se il numero di cicli è noto oppure no. Essendo noto il numero di termini, ossia 90, è noto anche il numero di cicli, per cui si dovrà utilizzare un ciclo `for`. Un grossolano diagramma potrebbe quindi essere il seguente:

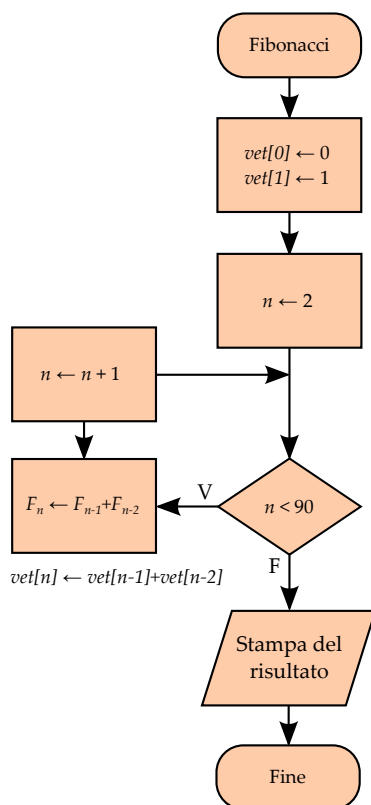



Figura 9.4: Successione di Fibonacci

Il primo blocco ha il compito di inizializzare i primi due elementi del vettore con i primi due valori della successione, ossia 0 e 1. Il secondo blocco rappresenta l'inizializzazione dell'indice, che viene posto a due, dato che l'iterazione riguarda solamente gli elementi dal terzo compreso in poi, come si evince dalla relazione 9.1 a pagina 255. Il cuore del diagramma e il blocco che solitamente richiede anche maggior sforzo di astrazione da parte dello studente è la sequenza posta nel corpo dell'iterazione. Tale blocco indica l'esecuzione

[3]Lo studente che si dovesse trovare spiazzato di fronte a Cartesio ed il metodo che il matematico francese propose, può trovare adeguata informazione nel documento *Problem Solving Strategies* dello stesso autore della presente.

della seguente azione:

$$F_n \leftarrow F_{n-1} + F_{n-2} \tag{9.3}$$
$$vet[n] = vet[n-1] + vet[n-2]; \tag{9.4}$$

L'espressione 9.3 è la rappresentazione matematica della parte ricorsiva della successione, mentre l'espressione 9.4 rappresenta l'equivalente assegnazione in linguaggio C. E' bene spendere qualche parola di commento per quest'ultima.

Si supponga che ad un certo punto dell'esecuzione del diagramma di fig. 12.1 l'indice assuma il valore $n = 10$. L'espressione di assegnazione 9.4, per sostituzione, diventerebbe:

$$vet[10] = vet[9] + vet[8]; \tag{9.5}$$

indicando che all'elemento avente indice 10 viene assegnato il valore dato dalla somma degli elementi aventi indice 9 e 8, che è l'esatta interpretazione della successione di Fibonacci.

Un possibile codice corrispondente al diagramma di flusso già visto ma *senza la stampa del risultato* potrebbe essere:

Listing 9.2: Successione di Fibonacci senza stampa

```
//Inizializzazione dei primi due elementi
//fuori dal ciclo. Sarebbe stato possibile
//porre detta inizializzazione nel for.
vet[0] = 0;
vet[1] = 1;

//Ciclo di calcolo degli elementi seguenti
for(n=2; n<90; n++)
   vet[n] = vet[n-1]+vet[n-2];
```

Una versione un po' più criptica e più concisa del codice precedente potrebbe essere:

Listing 9.3: Seconda versione di Fibonacci senza stampa

```
//Ciclo di calcolo della successione di Fibonacci
for(vet[0]=0, vet[1]=1, n=2; n<90; n++)
   vet[n] = vet[n-1]+vet[n-2];
```

Infine la stampa del risultato. Si deve ricordare che non si può stampare un vettore[4], per cui è necessario stampare ogni singolo elemento del vettore:

Listing 9.4: Versione definitiva di Fibonacci con stampa

```
//Ciclo di calcolo della successione di Fibonacci
for(vet[0]=0, vet[1]=1, n=2; n<90; n++)
   vet[n] = vet[n-1]+vet[n-2];

//Stampa del risultato
printf("I␣primi␣90␣termini␣della␣successione");
printf("␣di␣Fibonacci␣sono:\n");
for(n=0; n<90; n++)
   printf("%d\n", vet[n]);
```

[4]Anche in questo caso si vedrà, nelle prossime sezioni, un'eccezione relativa ad un particolare tipo di vettore: la stringa.

Si noti che stavolta il ciclo `for` inizializza l'indice a 0, in modo da stampare tutti gli elementi fra 0 e 89 compresi.

Si consiglia lo studente di riflettere sull'espressione 9.4 a fronte. Un concetto o una formula spiegata da un insegnante appare sempre semplice e banale. Ma quello che si vuole promuovere nello studente non è una qualche abilità imitativa ma una solida capacità di pensiero critico e autonomo. A tal fine serve la riflessione personale e una montagna di "perché?".

Esercizio - ◊◊⧫ Il Crivello di Eratostene

Uno dei metodi più geniali per individuare tutti i numeri primi in un intervallo dato è il Crivello di Eratostene. L'algoritmo è attribuito a Eratostene da Cirene (275-195 a.C.). Questi fu matematico, astronomo geografo, poeta e bibliotecario della biblioteca di Alessandria. E' famoso soprattutto per essere stato il primo a misurare la circonferenza della terra con ottima approssimazione. L'idea del Crivello è molto semplice: dato un numero primo, tutti i suoi multipli non sono primi, per cui è sufficiente *setacciare* (crivello = setaccio) i multipli di ciascun numero e trattenere solo quelli che non sono multipli di nessun numero. Per maggior semplicità, dato un vettore di n elementi, si fornisce il seguente algoritmo:

1. Segna come primi tutti gli elementi del vettore;
2. Per tutti gli elementi m compresi fra 2 e $n/2$ esegui quanto espresso nel passo 3;
3. Per tutti gli elementi f compresi fra m e n/m esegui quanto espresso nel passo 4;
4. Segna l'elemento indicato dal prodotto $f \cdot m$ come numero non primo;
5. Stampa gli indici degli elementi segnati come primi.

Si chiede di fornire il diagramma di flusso e il codice di detto algoritmo. Si invita lo studente a non leggere la soluzione finché non abbia prodotto quanto richiesto. Lo studente *non perda l'occasione* di cimentarsi con questo algoritmo, che richiede numerose riflessioni a volte anche semplicemente sui singoli termini. Dopo aver letto la soluzione, tale occasione non si presenterà più.

Soluzione

Come al solito, ci si dovrebbe chiedere se il testo è chiaro, oppure se vi sono delle ambiguità da chiarire. Storicamente, il presente esercizio ha rappresentato un ostacolo per molti studenti. L'algoritmo non si presenta per nulla semplice e tutto sommato potrebbe non essere chiarissimo nemmeno perché dovrebbe funzionare. Si propone, quindi, una riflessione sull'idea che ebbe Eratostene.

Dato un numero primo, tutti i suoi multipli non sono primi. Si supponga la sequente sequenza:

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Il primo numero della sequenza è 2, che è primo. Eratostene propone di cancellare dalla lista tutti i multipli di 2. Si ottiene la seguente sequenza:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15

Analogamente si potrebbe procedere con il 3, che non ha divisori a lui precedenti:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~

Alla fine del processo (di setacciatura) restano solamente i numeri primi. In base a questa idea si è proposto l'algoritmo precedentemente descritto.

Anche quest'ultimo, però, presenta qualche spunto di riflessione. Partiamo dall'inizio: "Sia dato un vettore di n elementi". E' ragionevole sostenere che si sia in presenza di una definizione di vettore. Più criptica è la frase seguente, riferita al primo *step* dell'algoritmo: "Segna come primi tutti gli elementi del vettore". L'autore confessa di aver volutamente utilizzato una terminologia poco informatica. Si tratta di interpretare cosa significhi "segnare un elemento del vettore". L'interpretazione non è di grande difficoltà: un elemento di un vettore si può solo "scrivere" o "leggere". Utilizzare altri verbi rappresenta licenza più o meno poetica, ma dovrà esserci, alla fine, sempre una convergenza verso uno dei due appena nominati. Da questo punto di vista "segnare" può essere affiancato solamente a "scrivere", quindi si chiede di inizializzare come primi tutti gli elementi del vettore. Si tratta di codificare detta informazione in qualche modo. Il modo più semplice è definire un vettore di tipo `char` e di scrivere in ciascun elemento il valore 1, ossia il valore di verità *true*. Graficamente, se $n = 16$, si ha quanto indicato in figura:

Figura 9.5: Crivello: Un vettore di 16 elementi

Come si potrebbe interpretare un vettore così formato? Il modo più semplice, tra l'altro confermato da quanto si afferma negli *step* seguenti dell'algoritmo, di interpretare la fig. 9.5 è sicuramente il seguente: l'indice rappresenta il numero, che può essere primo o meno, ed il contenuto del singolo elemento rappresenta la primalità o meno del relativo numero. Più precisamente se il contenuto dell'elemento è 1 tale valore indica che l'indice correlato è primo, se, ad esempio, l'elemento vale 0 (*false*) il correlato indice non è primo. Il vettore viene inizializzato in modo che ciascun elemento indichi che il correlato indice sia primo. Naturalmente ciò è falso, ma l'inizializzazione è solo il primo passo dell'algoritmo. Sicuramente, la prima cosa da fare prima di procedere con l'algoritmo, sarà di azzerare, ossia rendere *non primi* i primi due elementi del vettore, come indicato nella sottostante figura.

Figura 9.6: Crivello: Il vettore prima dell'esecuzione dell'algoritmo

Ciò fatto, si tratta di interpretare i successivi passi dell'algoritmo. I passi 2 e 3 sono piuttosto simili sia nell'*incipit* quanto nel verbo. In entrambi i casi si chiede di "eseguire" qualcosa "per tutti gli elementi ... compresi fra ... e ...". Il che significa che l'azione da eseguire va espletata su un elemento, poi sul successivo e così via per l'intervallo dato.

Ciò significa, però, che i due passi prevedono l'uso di due iterazioni. E siccome il numero di cicli in entrambi i casi è calcolabile, si tratta di due cicli `for`.

Più precisamente, l'azione eseguita dalla prima iterazione (passo 2) è l'esecuzione della seconda iterazione (passo 3) e l'azione eseguita da quest'ultima è quella indicata al passo 4. L'ultimo passo è la stampa.

Alla luce di quanto detto si può incominciare a tracciare il diagramma di flusso. Lo si farà per gradi in modo da suddividere il problema in sottoproblemi più semplici. Una prima versione potrebbe prevedere l'inizializzazione, l'esecuzione del passo 2 e la stampa del risultato. Il corpo del ciclo potrebbe essere formato dal passo 3.

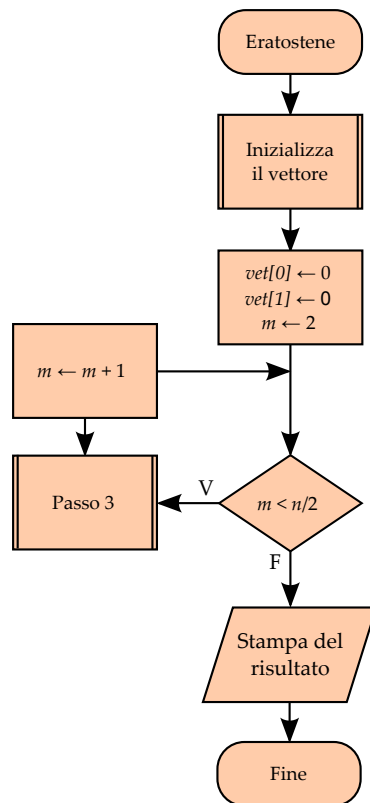



Figura 9.7: Crivello di Eratostene: prima versione

In fig. 9.7 è rappresentato quanto detto. Il primo blocco rappresenta l'inizializzazione del vettore, ove ciascun elemento viene posto a 1, ad indicare che detto elemento è correlato ad un indice che è primo. Il secondo blocco rappresenta l'inizializzazione del ciclo `for`, che prevede l'inizializzazione dell'indice, ma anche quella dei primi due elementi del vettore che, non avendo per indice due numeri primi (0 e 1 non sono primi per il Teorema fondamentale dell'Aritmetica), vanno azzerati.

Il blocco seguente è il test booleano e serve a determinare il numero di cicli: essi sono in numero di $n/2 - m + 1$. Supponendo $n = 100$ si ottiengono un numero di cicli pari a 49. Il corpo del ciclo è rappresentato dall'esecuzione del passo 3 ed infine dall'aggiornamento dell'indice m del ciclo.

L'ultimo blocco del diagramma rappresenta la stampa del vettore. In particolare, quest'ultima parte è già stata analizzata nell'esercizio precedente, per

cui non si commenterà oltre detto blocco.

Restano da definire i due blocchi relativi all'inizializzazione e al passo 3. La

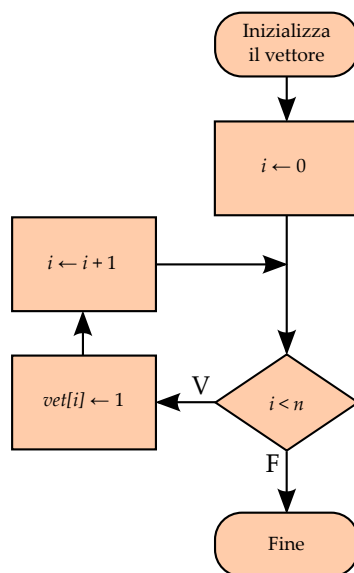



Figura 9.8: Crivello: Inizializzazione

prima azione è chiaramente un'iterazone a numero di cicli definito. Il numero dei cicli è n e la sequenza posta nel corpo del ciclo ha il compito di scrivere un 1 nell'elemento puntato. Ciò è illustrato in fig. 9.8. Rimane da definire l'ultimo blocco: l'esecuzione del passo 3. Si tratta del cuore vero e proprio

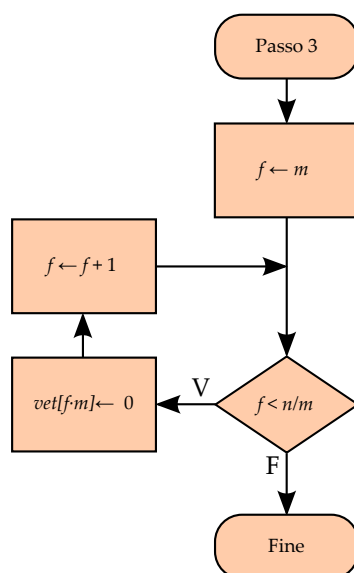



Figura 9.9: Crivello: Passo 3

dell’algoritmo. Si tratta di un’iterazione che esegue la propria sequenza per un numero di cicli pari a $n/m - m + 1$ (vedi fig. 9.9 nella pagina precedente).

Siccome l’idea di Eratostene è tutta concentrata nella sequenza eseguita la si ripropone per commentarla, in modo da fugare eventuali dubbi. La sequenza è la seguente:

$$vet[f \cdot m] \leftarrow 0 \tag{9.6}$$

Essa illustra l’assegnazione del valore 0 (che significa che l’indice dell’elemento non è primo) ad un elemento. Detto elemento ha indice $f \cdot m$, che è un prodotto. Il risultato di un prodotto, con $f, m > 1$, non può essere un numero primo per definizione. La scrittura del valore 0 avverrà, quindi, in un elemento con indice non primo. Al termine dell’algoritmo, il vettore apparirà così:

Figura 9.10: Crivello: Il vettore definitivo

A questo punto è possibile illustrare il codice, che apparirà come di seguito:

Listing 9.5: Crivello di Eratostene

```
//Si suppongono tutte le variabili intere
//Passo 1: Inizializzazione del vettore
for(i=0; i<n; i++)
   vet[i] = 1;

//Passo 2: Ciclo esterno
for(m=2; m<n/2; m++)
   //Passo 3: Ciclo interno
   for(f=m; f<n/m; f++)
      vet[f*m] = 0;     //Elemento non primo

//Stampa dei numeri primi
printf("I␣numeri␣primi␣compresi␣fra␣");
printf("2␣e␣%d␣sono:\n", n-1);
for(i=2; i<n; i++)
{
   if(vet[i]==1)
      printf("%d\n", i);
}
```

Una volta tanto, però, l’esercizio non finisce con la scrittura del codice. C’è un aspetto estremamente interessante dei due cicli che non si è affrontato prima per non appesantire eccessivamente l’analisi, ma che ora è opportuno fare.

Il corpo del ciclo interno esegue una assegnazione, che per comodità è stata riproposta in 9.6. In essa si vede che l’elemento con indice $f \cdot m$ viene posto a zero, indicando in tal modo che detto elemento è correlato ad un indice non primo. E’ però fondamentale che detto prodotto sia un valore che *identifichi un effettivo elemento del vettore*. Si supponga che il vettore sia composto da $n = 100$ elementi. Se il prodotto $f \cdot m$ è maggiore o uguale a n, si *indica un elemento non esistente del vettore* e ciò va assolutamente evitato. Dobbiamo quindi accertarci che il suddetto prodotto sia sempre minore di n.

Siccome il prodotto viene eseguito con gli indici dei due cicli, ci si deve chiedere quale sia il valore che ciascuno di essi assume durante l'esecuzione dell'algoritmo. Sicuramente si dovrà rispettare la seguente relazione:

$$f \cdot m < n \tag{9.7}$$

altrimenti si indicherebbe un elemento posto fuori dal vettore. Siccome $m \geq 0$ si ha:

$$f < n/m \tag{9.8}$$

che è esattamente la condizione booleana che viene testata nel secondo ciclo. Inoltre, siccome $f, m \geq 2$ si può riscrivere la 9.8 quando f assume il valore minimo, per stabilire quale debba essere il valore massimo di m:

$$2 < n/m \tag{9.9}$$

$$m < n/2 \tag{9.10}$$

ed in questo caso la 9.10 esprime la condizione booleana che viene testata nel primo ciclo. In tal modo abbiamo giustificato a posteriori le condizioni booleane dei due cicli.

Si invita lo studente a riflettere con attenzione su dette due espressioni dato che, statisticamente, rappresentano frequente motivo di inciampo per i giovani programmatori.

9.4 Un vettore particolare: la stringa

Un particolare tipo di vettore è il vettore di caratteri. Tipicamente esso viene utilizzato per visualizzare delle frasi o delle parole, che possono cambiare a seconda del contesto. Per tale motivo si è provveduto a poter fornire i vettori di caratteri, se lo si ritiene necessario, di apposito *terminatore*, che indicasse la fine effettiva del testo. Si fornisce, quindi la seguente

Definizione 13 (Stringa)**.**
La stringa è un vettore di caratteri dotato di terminatore.

Si veda, a tal proposito, la figura sottostante.

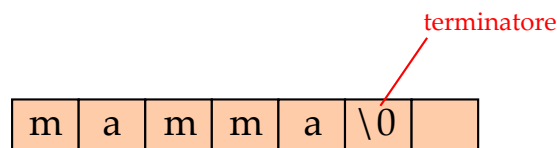



Figura 9.11: Esempio di stringa

Si nota che il vettore è formato da 7 elementi, ma che solo 6 sono effettivamente usati. Il settimo elemento contiene un carattere non noto, per cui non lo si è rappresentato. Quello che è importante notare è dato dal fatto che a stabilire la *effettiva* lunghezza della stringa è il *terminatore* (carattere `\0`, ovvero 0 o `NULL`), fermo restando, naturalmente, che il vettore sia uguale o più lungo della stringa, terminatore compreso.

Di seguito, per fissare il concetto, si dà l'esempio di definizione di vettore di caratteri e di definizione di stringa:

Listing 9.6: Vettori e stringhe

```
char vet[] = {'m','a','m','m','a'}; //Vettore
char stra[] = "mamma";               //Stringa
char strb[6] = "pluto";              //Stringa
char strc[20] = "pippo";             //Vettore abbondante
```

La prima è una definizione di vettore di caratteri, mentre la seconda è una definizione di stringa. La seconda differisce dalla prima perché contiene un terminatore in sesta posizione, dopo la seconda 'a'. Il vettore di caratteri, invece, è privo di terminatore. La terza definizione è equivalente alla precedente: viene solo esplicitata la effettiva lunghezza del vettore (non della stringa, si badi, che non conta il terminatore). L'ultima definizione è un po' abbondante e può tornare utile se si prevede la modifica della stringa.

Si sottolinea ancora che il terminatore limita la *effettiva* lunghezza della stringa, ma che il vettore può essere (e solitamente lo è) più lungo. Tale evenienza è frequente quando la stringa viene modificata *run time* per cui, al variare della stringa va spostato anche il terminatore. Il vettore di caratteri che contiene la stringa dovrà essere dimensionato in modo tale da contenere la massima stringa compreso il terminatore.

9.4.1 *Input/Output* di una stringa

La stringa costituisce un'eccezione nel panorama dei vettori, potendo essa, in certi casi, essere sia letta che scritta nel suo insieme e non solamente elemento per elemento. Si veda a tal proposito il seguente codice:

Listing 9.7: I/O di stringhe

```
char str[20];    //Allocazione di vettore di caratteri

scanf("%s", str);                   //Si digita "pluto"
printf("Si␣e'␣digitato:␣%s", str); //Stampa "pluto"
```

Inizialmente la (futura) stringa viene definita come vettore di caratteri. La dimensione del vettore viene scelta in modo tale da poter "contenere" la stringa voluta/immaginata compreso di terminatore.

Mediante la `scanf` è possibile digitare *tutta* la stringa e fornire l'invio da tastiera solo dopo averla digitata completamente. La funzione `scanf` provvede automaticamente ad aggiungere il terminatore a fine stringa *se il carattere di conversione è* "%s". Quindi se si digita `pluto` in realtà nel vettore verrà memorizzato `pluto\0`. Si noti che `\0` è un solo carattere e corrisponde allo zero o `NULL`, ovvero ad un carattere con tutti gli otto bit a zero. Si noti anche che, essendo la stringa un vettore, non si deve porre l'operatore di indirizzamento davanti alla stessa nella `scanf`.

Anche in fase di *output* non è necessario procedere alla stampa elemento per elemento se il carattere di conversione è "%s". La `printf` provvede a stampare tutti gli elementi della stringa fino al terminatore, che non verrà stampato.

9.4.2 Lettura/scrittura di una stringa

Al di fuori delle funzioni di *Input/Output* o al di fuori dell'inizializzazione in sede di definizione di variabile non è possibile leggere o scrivere le stringhe nel loro insieme: è sempre necessario operare elemento per elemento.

La lettura di una stringa dovrà probabilmente essere intrapresa mediante un ciclo `while`, dato che si deve ipotizzare l'eventualità che la stringa sia vuota. Si utilizza il termine "probabilmente" perché l'uso di un ciclo piuttosto che un altro dipende dal contesto. Se la stringa va solamente letta, ad esempio al fine di una sua interpretazione o codifica, il ciclo più indicato è il `while`.

Un'azione, invece, di altra natura, ma frequente, è la copia di una stringa. In questo caso il ciclo più adeguato è il `do..while`, dato che almeno un carattere, cioè il terminatore, va sempre copiato, come si evince nel codice seguente:

Listing 9.8: Copia di una stringa

```
char dest[20];           //Stringa destinazione
char source[] = "Pippo"; //Stringa sorgente
char* d;                 //Puntatori usati...
char* s;                 //...durante la copia

//Ciclo di copia.
s = source;
d = dest;
do
   *d++=*s;
while(*s++!='\0');
```

La stringa viene copiata carattere per carattere e quando si incontra il terminatore, che è già stato copiato, si esce dal ciclo. Si noti, però, che frequentemente a tal fine viene usata un'apposita libreria: la `string.h`. Mediante essa è possibile copiare, concatenare, eseguire ricerche, ecc. utilizzando stringhe. Nelle presenti pagine, per questioni di natura didattica si preferisce non utilizzare dette, tra l'altro utilissime, librerie.

9.4.3 Modifica di una stringa

Nella sezione precedente si è evidenziato un esempio di copia di stringa, che rappresenta un classico. Un'altra azione piuttosto frequente è la concatenazione di stringa, ovvero l'azione di unione di due stringhe a formarne una sola. Se si presenta tale eventualità ci si deve assicurare che il vettore di caratteri definito sia sufficientemente grande per contenere la stringa finale.

Si supponga di aver definito un vettore di caratteri `str` di 20 elementi e che detto vettore contenga la stringa "topo". Si supponga una seconda stringa `spider` contenente la parola "ragno". Si vuole concatenare la seconda stringa nella prima a formare la parola "toporagno".

Il codice potrebbe assomigliare al seguente:

Listing 9.9: Concatenazione di stringa

```
//Il seguente codice mostra un esempio di
//concatenazione di stringa. La stringa di
//partenza e' "topo", quella definitiva e'
//"toporagno".
```

```
char* s;
char* d;

s = spider;      //Puntatore sorgente
d = str;         //Puntatore destinazione

//Cerca il terminatore della prima stringa.
//Si usa un ciclo while per fermarsi esattamente
//sul terminatore
while(*++d!='\0');

//Concatena la seconda stringa. Si noti
//che il terminatore viene sovrascritto
//con la lettera 'r' e spostato a fine
//stringa.
do
   *d++ = *s;
while(*s++!='\0');
```

Si noti la particolare sintassi del primo ciclo `while`.

9.4.4 Esempio d'uso di stringhe

Il seguente è il solito esercizio relativo all'argomento trattato. Si coglie l'occasione per proporre un programma "riassuntivo" e un po' più complicato degli esercizi appena visti. In particolare, si tratta di un modo un po' diverso di calcolare il valore di π con un'approssimazione di una dozzina, o poco più, di cifre.

Esercizio - ◊◊♦ Ave o Roma

"Ave o Roma o Madre gagliarda di latine virtù che tanto luminoso splendore prodiga spargesti con la tua saggezza".

Questa frase serviva agli studenti di qualche anno fa per ricordarsi le prime cifre decimali del π (pigreco). La lunghezza di ogni parola della frase rappresenta una cifra del π, come evidenziato in figura:

Ave o Roma o Madre gagliarda di latine...
3 . 1 4 1 5 9 2 6

Figura 9.12: Ave o Roma...

Data la suddetta frase definita come stringa, si chiede di tracciare il diagramma di flusso e scrivere il codice in linguaggio C che"ricostruisca" il valore di π nascosto in detta frase e lo stampi.

Soluzione

Il valore che si ottiene scrivendo il numero di cifre di ciascuna parola, dopo aver posto una virgola (o punto) dopo la prima cifra è il seguente:

3.141592653589793238

Le cifre vanno "estratte" leggendo le lettere delle singole parole che formano la frase citata. Si dovrà pertanto utilizzare un ciclo e stabilire quale ciclo usare. Si potrebbe giustificare uno qualsiasi dei tre cicli, in dipendenza del punto di vista adottato: si potrebbero contare le lettere della frase e utilizzare un ciclo `for`, però non sembra sportivo. Sembra più corretto optare per un'iterazione a numero di cicli non definito a priori ed in tal caso, fra i due, pare più appropriato il ciclo `do..while`. Il diagramma potrebbe essere il seguente:

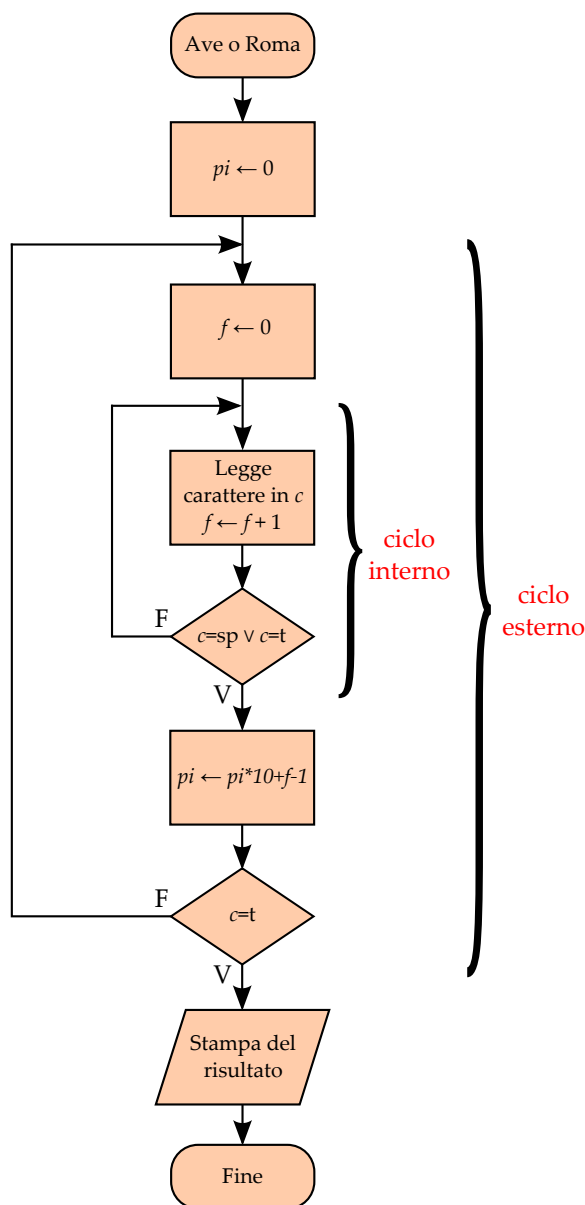



Figura 9.13: Diagramma estrazione π

Si potrebbe procedere a commentare il diagramma se esso non avesse un

evidente difetto: il numero ottenuto alla fine del processo è un numero intero di 19 cifre, ossia 3141592653589793238, che non ha nulla a che fare con il π, se non le cifre in comune. Si propone quindi, subito, una variante al diagramma per ristabilire un valore più accettabile di π. La variante è tratteggiata:

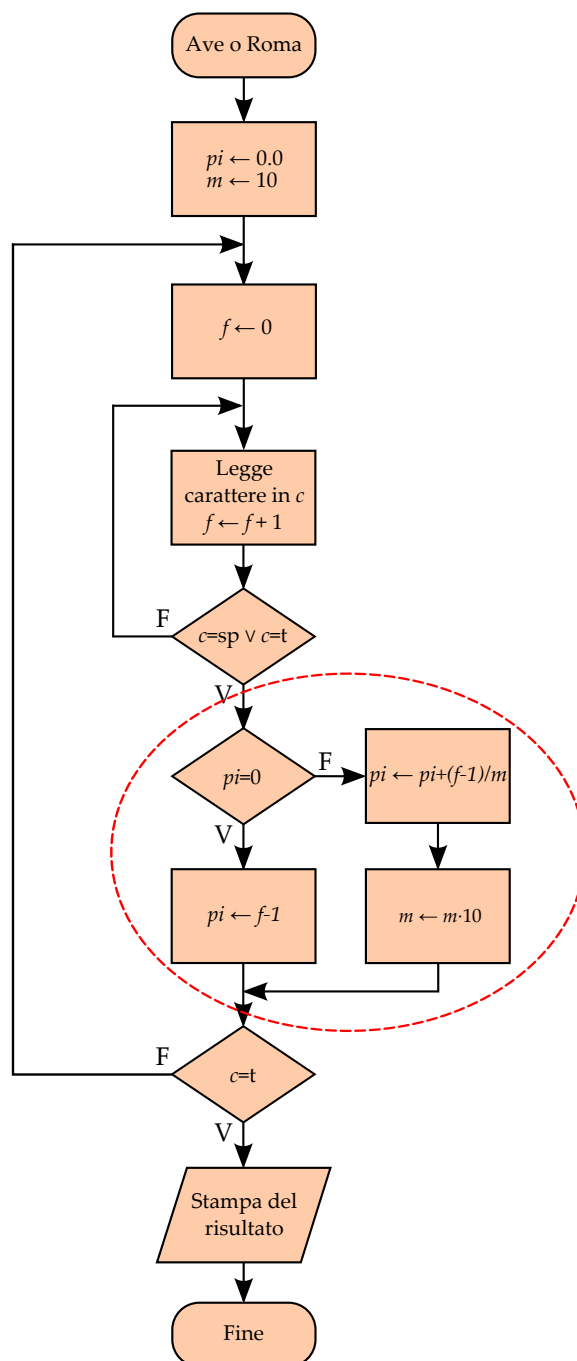



Figura 9.14: Diagramma estrazione π (versione definitiva)

Il primo blocco contiene l'inizializzazione del risultato `pi` e del divisore `m`, mentre il primo blocco del ciclo esterno prevede l'azzeramento del contatore di lettere `f` della singola parola. Il primo blocco del ciclo interno contiene la lettura del carattere corrente della stringa e l'incremento del contatore di lettere. Non è importante stabilire quale debba essere la successione delle due azioni.

Dette azioni vengono ripetute finché viene letto uno spazio oppure il terminatore di stringa. Nel primo caso si è rilevato la fine della parola e nel secondo caso la fine della parola e della stringa.

Quando viene identificato lo spazio, significa che la parola è terminata e che il contatore `f` contiene il numero di lettere della parola appena letta *più lo spazio*. Se dunque la parola era composta da 6 lettere, si avrà `f=7`.

Usciti dal primo ciclo si deve valutare se la parola di cui si è appena contato le lettere è la prima oppure no. Ciò è facilmente deducibile valutando il valore del risultato parziale: se vale 0 si è sicuramente in presenza della prima parola/cifra. In tal caso è sufficiente sommare il numero di lettere appena calcolato e presente in `f` decrementato di uno.

In caso contrario si sta calcolando la parte decimale del π per cui la cifra calcolata, dopo essere stata adeguatamente decrementata per i motivi già visti, va divisa per il divisore `m` e sommata al risultato parziale. Nel blocco seguente il divisore viene moltiplicato per 10 dato che il peso della prossima cifra sarà di un ordine di grandezza inferiore.

Ciò fatto si controlla se si è arrivati al termine della stringa, ovvero se l'ultimo carattere letto era il terminatore, altrimenti si ricomincia dall'inizio del ciclo esterno.

L'ultimo blocco è dato dalla stampa del risultato, ossia `pi`.

Un esempio di codice riferito al diagramma di fig. 9.14 potrebbe essere il seguente:

Listing 9.10: Estrazione π

```
//Per motivi grafici non si assegna a str tutta la stringa. Si
//deve supporre che nel programma cio' avvenga. Inoltre, si
//tolgono anche tutti gli accenti dalla detta stringa.
char str[]="Ave␣o␣Roma␣o␣Madre␣gagliarda␣di␣latine␣virtu...";

//Microsoft non supporta piu' il dato in virgola mobile a 80 bit
//equiparando la precisione del tipo long double a quella del
//double. Cio' implica che non si visualizzeranno ttutte le 18
//cifre decimali richieste, ma solamente 15.
double pi=0.0;          //Risultato parziale
long long int m=10;     //Divisore
int i=0;                //Indice stringa
int f;                  //Contatore di lettere
char c;                 //Carattere attualmente letto

//Ciclo esterno. Detto ciclo si occupa di processare le singole
//parole.
do
{
   //Azzera il contatore di lettere prima di leggere le lettere
   //della prossima parola.
   f=0;
```

```
    //Ciclo interno. Si leggono e contano le singole lettere
    //della parola corrente.
    do
    {
       c=str[i++];          //Legge il carattere
       f++;                 //Aggiorna il contatore di lettere
    }while((c!=' ')&&(c!='\0'));

    //Valuta se si deve aggiornare la parte intera o la parte
    //decimale del pigreco. Cio' si puo' dedurre valutando il
    //valore di pi.
    if(pi==0)
       pi=f-1;              //Aggiorna la parte intera
    else
    {
       pi+=(f-1.0)/m;       //Aggiorna la parte decimale e...
       m*=10;               //...il divisore
    }
}while(c!='\0');

//Stampa il risultato
printf("Il pigreco con 15 cifre decimali vale %1.15f", pi);
```

Il programma è praticamente identico al diagramma di flusso, per cui i commenti saranno ridotti al limite. Le uniche note sono relative alle condizioni booleane che sono state cambiate, al fatto che l'aggiornamento della parte decimale avviene con divisione decimale e al fatto che il divisore deve essere di tipo `long long int` dato il corposo numero di cifre.

9.5 Vettori multidimensionali

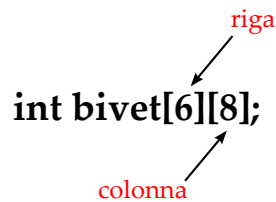
Il C, come gli altri linguaggi ad alto livello, permette la creazione di vettori multidimensionali. Un vettore bidimensionale ha la forma geometrica di una tabella (vedi fig. 9.15), ove il singolo elemento è identificato da una riga e da una colonna. Un vettore tridimensionale ha la forma geometrica di un cubo,

0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35

Figura 9.15: Vettore bidimensionale

il cui singolo elemento è identificato dalle tre misure poste sugli assi X, Y e Z. Naturalmente non vi è limite al numero delle dimensioni e il fatto che aumentando il loro numero aumenti anche la difficoltà di rappresentarle graficamente non sarà argomento di discussione.

Esattamente come si usava nei vettori monodimensionali un indice per accedere al singolo elemento, nei vettori multidimensionali si useranno n indici per accedere al singolo elemento del vettore con n dimensioni.

A titolo esemplificativo, la tabella di fig. 9.15 nella pagina precedente è definita come segue:

Figura 9.16: Esempio di definizione di vettore bidimensionale

La prima dimensione indica il numero di righe della tabella e la seconda dimensione indica il numero di colonne. Naturalmente l'uso dei termini "riga" e "colonna" è solo una convenzione utile per riferirsi alla struttura grafica del vettore. In realtà, i singoli elementi vengono memorizzati uno di seguito all'altro, come se si trattasse di un vettore monodimensionale. Come ciò avvenga verrà illustrato tra breve.

Prima è utile spiegare come si possa accedere in lettura ed in scrittura un elemento del vettore. Come in una tabella, esso va identificato attraverso la sua riga e colonna di appartenenza. Se si volesse accedere all'elemento della tabella di fig. 9.15, contenente il valore 18, appartenente alla colonna 6 e riga 3, quale sintassi si dovrebbe utilizzare? La sintassi sarebbe esattamente quella che intuitivamente si ipotizza:

Listing 9.11: [3] del vettore]Accesso all'elemento [6][3] del vettore

```
n = bivet[6][3]; //Lettura
bivet[6][3] = n; //Scrittura
```

Analogamente a quanto visto nel codice 9.1 a pagina 254 è possibile inizializzare anche i vettori multidimensionali. Entrambe le seguenti notazioni sono corrette:

Listing 9.12: Inizializzazione di vettori bidimensionali

```
int biveta[2][3] = {0,1,2,3,4,5};
int bivetb[][3] = {0,1,2,3,4,5};
```

Anche nella seconda definizione, infatti, il compilatore è perfettamente in grado di valutare la prima dimensione. Si ponga attenzione, però, che la seguente notazione *non* è corretta dal punto di vista sintattico:

Listing 9.13: Sintassi errata

```
int bivetc[2][] = {0,1,2,3,4,5}; //Sintassi errata
```

9.5.1 L'accesso agli elementi nei vettori multidimensionali

Le sintassi evidenziate nel codice 9.12 a fronte sono effettivamente piuttosto intuitive. Un po' meno intuitivo è valutare il valore di `n` nel seguente codice:

Listing 9.14: Quanto vale *n*?

```
int bivet[2][3] = {0,1,2,3,4,5};
n = bivet[1][2]; //Quanto vale n?
```

Bisognerebbe essere in grado di valutare con precisione l'organizzazione dei dati del vettore in memoria. Si è già detto nella sezione precedente che i dati vengono memorizzati uno di seguito all'altro. Nel caso presente, ad esempio, si potrebbe intuitivamente ricostruire una tabella $2x3$, chiamando "riga" il primo indice e "colonna" il secondo. Se però le dimensioni dovessero aumentare potrebbe risultare difficile definire l'effettiva organizzazione dei dati.

Cercando di chiarire il suddetto aspetto, si supponga il seguente vettore tridimensionale, sapendo che quanto verrà detto vale anche per dimensioni superiori:

int trivet[2][3][4] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23};

Figura 9.17: Inizializzazione di vettore tridimensionale

Se si volesse accedere all'elemento [1][0][2], quale valore numerico verrebbe letto? La risposta è data dalla seguente figura:

int trivet[2][3][4] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23};

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	1	1	1	1	2	2	2	2	0	0	0	0	1	1	1	1	2	2	2	2
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Figura 9.18: Posizionamento degli elementi del vettore

La notazione [2][3][4] indica che ci sono complessivamente $2x3x4 = 24$ elementi nel vettore. Il primo indice (2) stabilisce la presenza di *2 vettori bidimensionali*. Ciò significa (vedi fig. 9.18) che la prima metà dei dati, ossia la prima delle due tabelle bidimensionali, sarà indicata dal valore 0 di detto indice e la seconda metà dei dati dal valore 1.

Il secondo indice (3) stabilisce che ciascuna delle due parti precedentemente illustrate del vettore *è a sua volta suddivisa in tre ulteriori parti* (o meglio vettori), ciascuna/o delle quali formata da *4 elementi*.

Quindi la notazione [1][0][2] indica che l'elemento indicato risiede la seconda parte del vettore (1), più precisamente nel primo (0) dei tre vettori ivi presenti e che l'elemento effettivamente indicato in quest'ultimo vettore è il terzo (2): il valore contenuto in detto elemento è quindi 14. Si noti che è sufficiente leggere in verticale, dal basso vero l'alto, i tre indici posizionati sotto il valore 14.

Se la suddetta organizzazione non è chiara nella mente dello studente, egli rischia di accedere all'elemento errato nel vettore, con risultati probabilmente pessimi.

Si fa notare che, similmente a quanto detto per i vettori bidimensionali, è corretta anche la notazione `int trivet[][3][4] = {0,1,2,....`

9.6 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 9.

Capitolo 10

I puntatori

Or incominciàn le dolenti note
a farmisi sentire; or son venuto
là dove molto pianto mi percuote.

Inferno, V canto - Dante Alighieri

Come si intuisce dall'introduzione un po' tetra del presente capitolo, l'argomento non è dei più semplici. Anzi, si tratta sicuramente dell'argomento che lo studente più difficilmente assimila e fa proprio. E' quindi estremamente importante che lo studente studi il presente capitolo con particolarissima attenzione se vuole impadronirsene. L'argomento trattato è il puntatore, per cui si fornisce subito una sua definizione:

Definizione 14 (Puntatore)**.**
Il puntatore è una variabile contenente un indirizzo di memoria che indica dove inizia la struttura dati o la funzione da esso indirizzata.

Dunque, il puntatore è una variabile contenente un indirizzo di memoria. Rimane da stabilire a cosa potrebbe servire. Le destinazioni d'uso del puntatore sono molteplici: può essere usato per puntare a metà di una struttura dati piuttosto che al suo inizio, più o meno come si può fare con l'indice di un vettore; può essere usato per puntare una variabile senza doverne usare l'identificatore che la contraddistingue; frequentemente si usa passare il puntatore ad una struttura dati come argomento di una funzione,[1] soprattutto se la struttura dati è voluminosa, e così via.

In ogni caso si tratta di uno strumento potentissimo, soprattutto a livelli piuttosto avanzati di utilizzo del linguaggio C, che rende la programmazione molto più efficiente e flessibile. Purtroppo, però, è anche molto facile fare grandi danni usando detto strumento, per cui sarà d'obbligo molta prudenza.

[1]Sarà l'argomento del prossimo capitolo.

Quanto detto sui puntatori potrebbe essere illustrato graficamente nel seguente modo:

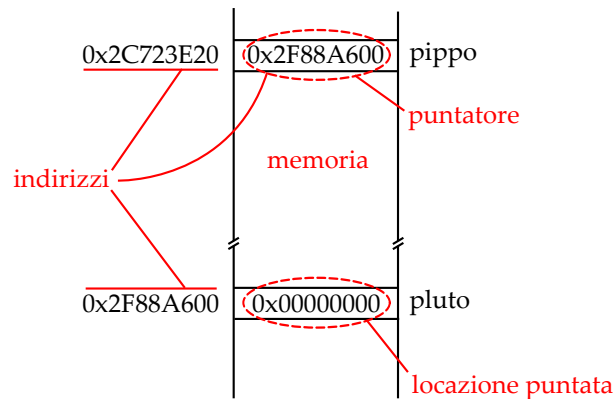



Figura 10.1: Locazione puntata da puntatore

In figura compare una rappresentazione grafica della memoria di un PC. Essa è rappresentata come un'enorme cassettiera[2] verticale dove, nel presente caso, sono stati evidenziati solamente due cassetti: quello riferito a `pippo` e quello riferito a `pluto`.

Nel disegno `pippo` va interpretato come puntatore. Ciò significa che il suo contenuto è un indirizzo, più precisamente l'indirizzo 0x2F88A600, che corrisponde all'indirizzo di inizio della variabile `pluto`, il cui contenuto è zero.

10.1 La definizione di puntatore

La definizione di puntatore si presta ad interessanti quanto utili riflessioni. Vediamone innanzi tutto la forma sintattica:

Listing 10.1: Definizione di puntatore

```
int* p; //Definizione di puntatore a intero
```

La prima cosa che colpisce è la presenza del simbolo "*" posto dopo l'identificatore di tipo. Si precisa subito che detto simbolo *non fa parte dell'identificatore stesso*, ma ha un valore semantico ben preciso, in quanto identifica e caratterizza la definizione di puntatore. Quindi, ogni qualvolta si rileva un asterisco all'interno di una definizione o dichiarazione, siamo sicuri che si è in presenza di un puntatore.

Come si intuisce dalla definizione evidenziata in 10.1, il puntatore è (quasi sempre) *tipizzato*, il che significa che si possono avere puntatori a intero, a numero in virgola mobile, a carattere, ecc., anche se si possono definire puntatori non tipizzati come nel seguente esempio e sui quali avremo modo di tornare:

Listing 10.2: Definizione di puntatore a void

```
void* p;        //Definizione di puntatore a void
```

[2]Nel disegno ogni cassetto "contiene" un'informazione di 32 bit. In realtà, solitamente, i singoli cassetti "contengono" informazioni di soli 8 bit. Ciò significa l'informazione 0x2F88A600 è dislocata su 4 cassetti adiacenti diversi.

10.2 Gli operatori relativi ai puntatori

Come si ha avuto modo di intravvedere nel capitolo 6 ed in particolare commentando la tabella 6.15 a pagina 187, il simbolo "*" è collocato sia fra operatori binari, con significato di operatore di moltiplicazione, sia fra gli opearatori unari. Analogamente, anche l'operatore "&" è collocato sia fra gli operatori binari, con significato di operatore di AND a bit, che fra gli operatori unari.

Il loro significato come operatore unario è strettamente legato ai puntatori e questo sarà l'argomento delle prossime due sezioni.

10.2.1 L'operatore di indirezione

L'operatore unario "*" (asterisco) è detto operatore di *indirezione* e permette di *accedere all'oggetto puntato*. Prima di approfondire l'argomento, si vuole subito essere chiari sulla terminologia da usare. E' frequentissimo sentir dire dagli allievi che "`*p` è un puntatore". Si tratta di un grossolano errore. Nel tentativo di impedire simili scivoloni agli studenti, si propongono le seguenti due definizioni, che sono semanticamente assolutamente identiche, pur presentando una piccola differenza sintattica:

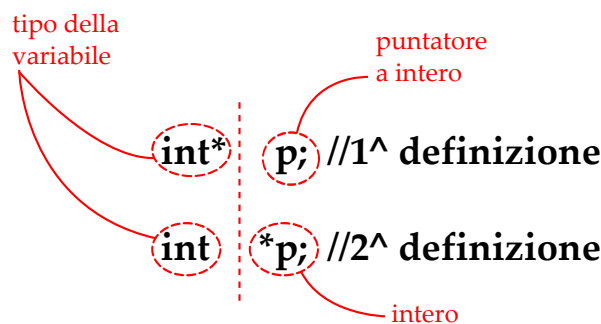



Figura 10.2: Stessa definizione, letture diverse

Le due definizioni separano il tipo dalla variabile in due modi sintatticamente diversi, ma dal punto di vista semantico, sono assolutamente identiche. Esse hanno però una particolarità: ci permettono di leggere senza errori le due strutture sintattiche.

Entrambe le definizioni sono attraversate da una linea tratteggiata, che "separa" il tipo, a sinistra del tratteggio, dalla variabile, posta alla destra del tratteggio. Per sapere "cos'è" l'oggetto posto alla destra della linea tratteggiata è sufficiente guardare cosa c'è alla sinistra della stessa. Quindi, dalla prima definizione si apprende che

`p` è un puntatore a intero

mentre dalla seconda che

`*p` è un intero.

Distinguere le due affermazioni senza esitazioni è assolutamente fondamentale se si vuole operare con i puntatori. Ciò permette anche di non fare errori sintattici o semantici in sede di scrittura di programmi.

10.2.2 L'operatore di indirizzamento

L'operatore unario "&" (*ampersand*[3]) è detto operatore di *indirizzamento* e permette di *estrarre l'indirizzo di inizio dell'area di memoria ove la variabile è allocata*. Con riferimento alla fig. 10.1 a pagina 278 se l'operatore & viene posto davanti alla variabile `pluto`, restituisce il valore 0x2F88A600, ovvero l'indirizzo d'inizio dell'area di memoria ove `pluto` è situato. Per essere più precisi, occupando `pluto` 4 byte (il che significa che potrebbe essere una variabile di tipo `int`), 0x2F88A600 è l'indirizzo del primo dei 4 byte che la variabile occupa.

10.2.3 Utilizzo dei due operatori

Le implicazioni palesi e nascoste connesse all'utilizzo dei due operatori relativi ai puntatori sono molte. Il miglior modo per porle in evidenza è fare alcuni esempi di codice e commentarli.

Listing 10.3: Esempi sui puntatori

```
int a, b=3;
int* c;   //1. c e' un puntatore a intero

a = b;    //2. Si sta copiando b in a
c = &b;   //3. Estrae l'indirizzo di b e lo pone in c
*c = 5;   //4. Si sta modificando b: b = 5 e a = 3
b = *c+a; //5. Equivalente ad b+=a
a = *c;   //6. a = 8
```

1. La definizione è relativa ad un puntatore a intero, lo si capisce dalla presenza nella definizione dell'asterisco. Si fa notare che come l'intero `a` contiene un valore a caso e non noto[4], analogamente anche il puntatore `c` punta una locazione di memoria a caso e non nota. Non inizializzare il puntatore può essere fonte di *crash* clamorosi in fase di *debugging*.
2. L'istruzione al punto due è assolutamente ovvia e conosciuta allo studente. Si vuole però porre in evidenza come, attraverso l'assegnazione, si stia *copiando* il valore di `b` in `a`. Si potrebbe pensare alla variabile `a` come alla fotocopia di `b`. Continuando l'analogia, modificando la copia, ossia `a`, l'originale, cioè `b`, rimane inalterato. Si tenga presente questa similitudine, perché tornerà utile.
3. Viene estratto l'indirizzo di allocazione della variabile `b` e assegnato al puntatore `c`. Si noti che da questo momento in poi, l'intero `*c` e l'intero `b`, non sono solo uguali, ma operano sulla stessa area di memoria. Riprendendo la similitudine della fotocopia, sia `*c` che `b` sono degli originali. Detto in altri termini se si modifica `*c` si modifica l'originale, ovvero anche `b`.
4. Quanto detto nel punto precedente appare ora evidente. Assegnando il valore 5 all'intero `*c` si modifica anche l'intero `b`, dato che `*c` e `b` operano sulla stessa area di memoria. Si noti che la copia `a` del vecchio valore di `b` è rimasta inalterata, dato che è stato modificato l'originale.

---

[3]La parola *ampersand* è la storpiatura delle parole inglesi e latine "and per sè and" ad indicare che il simbolo & (and) indica (è di per sé) l'operatore logico AND.

[4]Ciò non è sempre vero. Si veda la nota 6 a pag. 129

5. Ormai, le ultime due espressioni non dovrebbero creare difficoltà, se i punti precedenti risultano essere chiari. Siccome `*c` e `b` operano sulla stessa area di memoria, l'espressione indicata al punto cinque è assimilabile all'espressione `b += a;`.
6. Si esegue una nuova copia `a` di `b`, aggiornando in tal modo il valore di `a` al valore assunto da `b`, cioè 8.

Si ponga attenzione al fatto che il concetto di puntatore è estremamente infido. E' frequentissimo vedere studenti che credono di aver assimilato il concetto di puntatore inciampare clamorosamente e compiere errori, spesso gravi, in assoluta tranquillità. Per tali motivi si invita lo studente a studiare con estrema attenzione il presente argomento, approfondendolo con considerazioni personali e arriccendolo con riflessioni autonome.

10.3 Puntatori e vettori

Dal punto di vista sintattico vi sono degli aspetti interessanti da osservare quando si parla di vettori e puntatori e più precisamente di puntatori a vettori. Si fornisce a tal proposito la seguente

Definizione 15 (Puntatore a vettore)**.**
Il vettore è puntatore a se stesso, quindi l'identificatore del vettore è anche puntatore al vettore stesso.

Si è volutamente posticipato la suddetta definizione, perché mancava ancora il concetto di puntatore nel capitolo precedente. Si vedano a tal proposito i seguenti esempi posti nel codice 10.4:

Listing 10.4: Puntatori e vettori

```
int* p;
int vet[] = {2, 5, 0, 8, 7};

p = vet;         //1. Equivale a p = &vet[0]
vet[3]+=*p++;    //2. Cosa viene incrementato?
*p = 3;          //3. Qual e' l'elemento coinvolto?
p = &vet[2];     //4. E' corretto?
(*p)++;          //5. Cosa viene incrementato?
```

1. L'espressione relativa al punto uno è l'esemplificazione della definizione data. Si nota l'assenza dell'operatore di indirizzamento, perché assolutamente non pertinente. Infatti `p` e `vet` appartengono allo stesso tipo, essendo entrambi dei puntatori a intero. Nel commento si sottolinea che l'espressione `p=vet` è equivalente a `p=&vet[0]`. Non era nemmeno necessario sottolinearlo, dato che il puntatore è l'indirizzo di inizio dell'area di memoria ove la variabile puntata è allocata.
2. Per comprendere l'espressione `*p++` è bene dare una sbirciatina alla tabella 6.15 a pagina 187 e rinfrescare alcuni concetti. Il puntatore `p` è sottoposto all'azione di due operatori unari: l'operatore di indirezione e l'operatore di postincremento. Entrambi detti operatori hanno associatività da destra verso sinistra, il che significa che la variabile è *prima* sottoposta

all'azione dell'operatore posto più a destra e *poi* a quello posto più a sinistra. Quindi l'operatore di postincremento è riferito al puntatore e non all'oggetto puntato. Quindi l'azione eseguita dall'espressione commentata al punto due è la seguente: all'elemento 3 del vettore viene assegnato il risultato della somma fra il valore dello stesso elemento 3, ossia 8 e l'intero `*p`, ossia 2, dopodiché viene incrementato il puntatore che punterà all'elemento 1 del vettore.

3. All'elemento attualmente puntato da `p`, cioè all'elemento 2 del vettore, viene assegnato il valore 3. L'azione è legittima, dato che sia a destra che a sinistra dell'operatore di assegnazione si hanno due interi.
4. Anche in questo caso l'azione è corretta. Al puntatore `p` viene assegnato un nuovo indirizzo, ossia quello dell'elemento 2 del vettore, contenente attualmente 0. Si noti che in tal modo si può inizializzare un puntatore ad un qualunque punto di una struttura dati.
5. Vi sono delle similitudini con la sorgente dell'espressione due. Adesso, però, l'ordine dell'associatività viene sovvertito dalla presenza delle parentesi, per cui l'operatore di postincremento agisce non più sul puntatore, ma su ciò che è posto fra parentesi, ovvero l'oggetto puntati. Quindi viene incrementato il terzo elemento del vettore, che passa da 0 a 1. Il vettore diventa pertanto il seguente: {2, 3, 1, 10, 7}.

10.4 Aritmetica dei puntatori

L'espressione `vet[3]+=*p++;` vista nella sezione precedente merita un approfondimento. Si è detto, con molta *nonchalance*, che "viene incrementato il puntatore che punterà all'elemento 1 del vettore". In realtà la frase avrebbe dovuto smuovere qualche dubbio nella mente del lettore, essendo `p` un *puntatore a intero* ed essendo esso *incrementato*.

Si suppongano, ad esempio, i numeri interi 123456789 e 987654321, rappresentati in esadecimale rispettivamente con 0x075BCD15 e 0x3ADE68B1 e che essi vengano memorizzati in modalità *big endian*[5], come evidenziato in fig. 10.3.

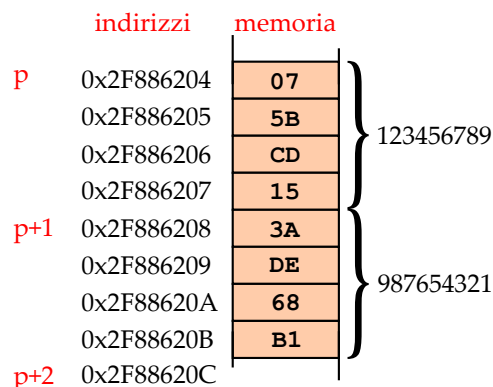



Figura 10.3: Aritmetica dei puntatori

[5]Per quanto riguarda i concetti di *big endian* e *little endian*, si veda l'appendice B.

La rappresentazione grafica fornita è quella tipica di due elementi adiacenti di un vettore. Se il numero 123456789 è allocato all'indirizzo indicato da `p`, nel caso presente 0x2F886204, l'incremento di `p` dovrebbe aggiornare il valore contenuto da `p` a 0x2F886205 e non a 0x2F886208. Invece viene aggiornato proprio a quest'ultimo valore. Il motivo risiede proprio nell'aritmetica usata dai puntatori.

Quando viene definito o dichiarato un puntatore, viene indicato anche il *tipo* associato al puntatore (puntatore a `int`, puntatore a `char`, ecc.), mediante il quale è possibile risalire allo spazio occupato dalla variabile puntata. Nel caso presente, siccome si è ipotizzato che i numeri fossero interi, cioè 4 byte per ciascun numero, incrementare il puntatore a intero significa *puntare il prossimo intero*, ovvero operare uno spiazzamento di 4 byte e non di 1.

Se il puntatore puntasse a una variabile di tipo `double`, che occupa 8 byte, incrementare il puntatore significherebbe aumentarne il valore di 8 unità.

Il seguente codice sviluppa attraverso alcuni esempi l'aritmetica dei puntatori:

Listing 10.5: Aritmetica dei puntatori

```
double d[] = {1.0, 2.0, 3.0, 4.0, 5.0};
int i[] = {100, 200, 300, 400, 500};
char c = {10, 20, 30, 40, 50};
double* pd;
int* pi;
char* pc;

//Gli indirizzi riportati in commento
//rappresentano un esempio
pd = d;
pi = i;
pc = c;

printf("pd␣␣␣=␣0x%x\n", pd);     //pd   = 0x1DFDEC
pd++;
printf("pd+1␣=␣0x%x\n", pd);     //pd+1 = 0x1DFDF4
pd+=2;
printf("pd+3␣=␣0x%x\n\n", pd);   //pd+3 = 0x1DFE04

printf("pi␣␣␣=␣0x%x\n", pi);     //pi   = 0x1DFDD0
pi++;
printf("pi+1␣=␣0x%x\n", pi);     //pi+1 = 0x1DFDD4
pi+=2;
printf("pi+3␣=␣0x%x\n\n", pi);   //pi+3 = 0x1DFDDC

printf("pc␣␣␣=␣0x%x\n", pc);     //pc   = 0x1DFDC0
pc++;
printf("pc+1␣=␣0x%x\n", pc);     //pc+1 = 0x1DFDC1
pc+=2;
printf("pc+3␣=␣0x%x\n", pc);     //pc+4 = 0x1DFDC3
```

Dal codice si evince che vengono definiti e inizializzati tre vettori di 5 elementi ciascuno: un vettore di `double`, nel quale ciascun elemento occupa 64 bit, ovvero 8 byte; un vettore di `int` ove ciascun elemento occupa 4 byte; un vettore di `char`, nel quale ciascun elemento occupa un solo byte.

Dopo la sezione di definizione vengono inizializzati i tre puntatori: pd (puntatore a numero in virgola mobile), pi (puntatore a intero) e pc (puntatore a carattere). Si noti come non viene mai usato l'operatore di indirizzamento per estrarre l'indirizzo, dato che l'identificato del vettore è già puntatore a se stesso.

Seguono poi tre gruppi di stampe, uno per ciascun puntatore. Nel primo gruppo viene stampato l'indirizzo puntato da pd, quello puntato da pd+1 e quello puntato da pd+2. Si è supposto che il primo indirizzo fosse 0x1DFDEC, per cui l'indirizzo puntato da pd+1 deve essere 0x1DFDF4, ossia di 8 unità maggiore. L'indirizzo puntato da pd+3 è di 24 unità più grande dell'indirizzo di inizio vettore.

Nel secondo gruppo gli spiazzamenti rispetto al puntatore d'inizio sono gli stessi: pi, pi+1 e pi+3. Gli indirizzi hanno però spiazzamenti diversi. Ciò è dovuto al fatto che il tipo int occupa solamente 4 byte, ovvero la metà rispetto ad un double. Gli indirizzi sono 0x1DFDD0, 0x1DFDD4 (+4) e 0x1DFDDC (+12).

Nell'ultimo gruppo gli spiazzamenti relativi agli indirizzi cambiano ancora, dato che stavolta il puntatore pc è un puntatore a carattere. In tal caso l'aritmetica dei puntatori e l'aritmetica "solita" coincidono, occupando il tipo char un solo byte. Gli indirizzo diventano 0x1DFDC0, 0x1DFDC1 e 0x1DFDC3.

10.5 Un'osservazione importante

Attraverso l'aritmetica dei puntatori si è riusciti a stabilire molte analogie fra puntatori e vettori, individuando, addirittura, un legame molto stretto fra l'identificatore del vettore ed il puntatore al vettore stesso.

Vi è però un'osservazione molto importante da fare al fine di evitare grossolani errori: *il puntatore è una variabile, mentre l'identificatore di un vettore non è una variabile*.

Quindi sono corrette le seguenti espressioni:

Listing 10.6: Espressioni corrette riguardanti vettori

```
int* p;
int vet[10];

scanf("%d", vet);   //Non serve l'ampersand
p = vet;            //Espressione utilizzante puntatori
p = &vet[3];        //L'ampersand e' necessario
*(vet+1) = *(vet+2);//vet non cambia valore
```

Sono invece errate le seguenti espressioni:

Listing 10.7: Espressioni errate riguardanti vettori

```
int* p;
int vet[10];

vet = p;            //Errore: vet non e' una variabile
vet++;              //Errore: vet non e' una variabile
*(vet+1)++;         //Errore: vet non e' una variabile
```

Nelle espressioni relative al codice 10.6 `vet` mantiene sempre il suo valore, ossia l'indirizzo di inizio dell'area di memoria riservata al vettore. Nelle espressioni relative al codice 10.7, invece, si tenta di far assumere a `vet` valori diversi da quelli che le competono, si cerca cioè di farlo diventare una variabile, il che non è possibile.

10.6 Ancora Fibonacci

Può essere interessante valutare come cambia il codice di una porzione di programma quando invece di usare la sintassi tipica dei vettori si usa quella dei puntatori. Di seguito si ripropone l'esercizio sulla successione di Fibonacci visto nel capitolo precedente, ma utilizzando per i calcoli i puntatori.

Esercizio - ◊◊◊ Fibonacci con i puntatori

Si chiede di riscrivere il codice in linguaggio C dell'algoritmo di Fibonacci utilizzando per i calcoli e per la stampa i puntatori.

Soluzione

Un esempio di come il codice possa essere riscritto utilizzando i puntatori invece della notazione dei vettori è dato di seguito. Si nota facilmente che non compaiono mai parentesi quadre, tipiche della sintassi dei vettori, se non nella sezione dichiarativa.

Listing 10.8: Fibonacci con i puntatori

```
long long int vet[90];
long long int *p;
int n;

//Ciclo di calcolo della successione di Fibonacci
p = vet;
for(*p=0, *(p+1)=1, n=2; n<90; n++)
   *(p+2) = *(p+1)+*p++;

//Stampa del risultato
p = vet;
printf("I␣primi␣90␣termini␣della␣successione");
printf("␣di␣Fibonacci␣sono:\n");
for(n=0; n<90; n++)
   printf("%d\n", *p++);
```

Si nota come si passi molto facilmente dalla sintassi dei vettori a quella dei puntatori. Unica nota: il valore dell'indirizzo d'inizio del vettore viene passato *ad una variabile* (`p=vet;`) in modo da poterla incrementare ad ogni ciclo. Prima della stampa si deve ripristinare detto valore dato che alla fine del ciclo di calcolo `p` punta alla fine del vettore.

10.7 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 10.

Capitolo 11

Le funzioni

Le funzioni sono un utilissimo strumento per semplificare la risoluzione dei problemi di programmazione. Permettono l'applicazione della strategia *divide et impera* isolando totalmente dal resto del programma una sua parte. Esse hanno lo stesso compito che hanno le *subroutines* nel diagramma a blocchi, con l'unica aggiunta di specificare formalmente quali siano gli argomenti che accetta per eseguire il processo ad essa richiesto.

Prima di introdurre alcuni concetti-chiave relativi alle funzioni si forniscono alcuni elementi terminologici, illustrando la struttura sintattica:

Figura 11.1: Struttura sintattica della funzione

La prima riga è formata da tre parti distinte e sempre presenti:

- il tipo di ritorno;
- il nome della funzione;
- gli argomenti o parametri della funzione.

Tale parte è detta *titolo della funzione*. Se essa è seguita da punto e virgola (;) la stessa parte prende il nome di *prototipo della funzione* ed ha funzioni prettamente dichiarative.

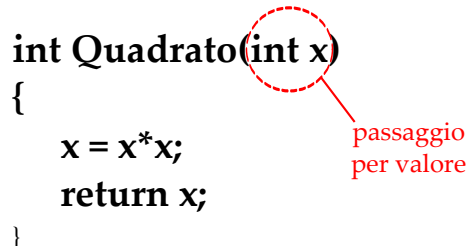
Il titolo della funzione è seguito dalla coppia di parentesi graffe che racchiudono il *corpo della funzione*, solitamente indentato rispetto alle graffe.

La funzione può ritornare un valore che essa ha elaborato. Il *tipo* di tale valore deve essere dichiarato nella prima parte del titolo della funzione. Se la funzione *non* ritorna alcun valore si deve indicare `void` nel tipo. All'argomento, data la complessità, verrà dedicata un'apposita sezione.

Il nome della funzione è indicato dall'identificatore dichiarato. Esso deve osservare le regole proprie degli identificatori ed indicate nella sezione 2.2.

Seguono il nome della funzione le parentesi tonde aperte e chiuse che racchiudono la *lista degli argomenti*. Non vi è limite teorico al numero degli argomenti di detta lista. Ciascun argomento va separato dal successivo dal separatore virgola (,) e va definito per tipo e nome, esattamente come nelle definizioni o dichiarazioni di variabili. La prossima sezione fornisce ulteriori dettagli sul passaggio degli argomenti alle funzioni.

11.1 Il passaggio degli argomenti

Il passaggio degli argomenti alla funzione costituisce un argomento estremamente importante dello studio delle funzioni, in quanto gli argomenti o parametri svolgono un compito fondamentale: costituiscono il collegamento della funzione con il mondo esterno. L'argomento è piuttosto delicato e merita adeguato approfondimento.

Uno dei concetti introdotti dalla programmazione strutturata è quello di *variabile locale*, contrapposto a quello di *variabile globale*. In maniera intuitiva ed approssimativa si potrebbe definire "locale" una variabile che esiste solo all'interno di una funzione: quando la funzione termina viene deallocata anche la variabile. La variabile "globale", per contrasto, potrebbe essere definita come una variabile utilizzabile in più funzioni e che non viene deallocata alla chiusura di una funzione. Definizioni più accurate verranno fornite quando si tratterà la *visibilità delle variabili*.

Tale differenziazione fra le variabili ha il compito di permettere una programmazione concettualmente più ordinata e una più semplice procedura di creazione delle librerie. Entrambe le azioni spossono diventare estremamente complesse se si fa uso solamente di variabili globali. Il *debugging* di una funzione, ad esempio, è molto più semplice se essa utilizza solamente variabili locali. La creazione di una libreria può diventare cervellotica con la presenza anche di una sola variabile globale.

Per tali motivi si tende ad utilizzare solamente variabili locali all'interno di una funzione, *passando* quei valori che necessariamente provvengono dall'esterno della funzione stessa. I modi di "passare" detti valori sono sostanzialmente due e verranno trattati nelle prossime due sezioni.

11.1.1 Il passaggio per valore

La sintassi relativa al passaggio per valore di un argomento ad una funzione è illustrato nella figura sottostante:

```
int Quadrato(int x)
{
    x = x*x;
    return x;
}
```

passaggio per valore

Figura 11.2: Passaggio per valore

Il passaggio per valore è caratterizzato dal fatto che la variabile passata viene *copiata* dalla funzione, che opera sempre sulla copia e mai sull'orginale (vedi la sezione 10.2.3). Se, all'interno della funzione la variabile passata per valore viene modificata, non viene modificato l'originale. Dal punto di vista sintattico la definizione di argomento è identica alla definizione di variabile.

Al fine di chiarire i concetti fin qui esposti si propone il seguente banale esempio. Si utilizza una ipotetica funzione `Quadrato` che restituisce il quadrato di un determinato valore. Naturalmente non è necessario scrivere una funzione per così poco, ma lo si fa per poter accentrare l'attenzione sui meccanismi di passaggio piuttosto che sul corpo della funzione.

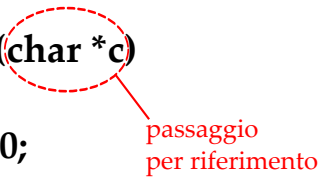
Listing 11.1: Esempio di passaggio per valore

```
int Quadrato(int x)
{
   x = x*x;      //L'argomento viene modificato
   return x;
}

void main(void)
{
   int a=2;      //Variabili di cui calcolare il quadrato
   int b=3;

   a = Quadrato(a);                //a=4
   a = Quadrato(2);                //a=4
   b = Quadrato(b);                //b=9
   a = Quadrato(a+b);              //a=169
   a = Quadrato(Quadrato(b/2));    //a=256
   a = Quadrato(32+Quadrato(b)-a); //a=?
}
```

Si noti quanto segue:

1. la funzione è scritta *prima* del `main` nel quale è utilizzata. Se non si fa uso della dichiarazione di prototipo, che verrà esaminata nelle prossime sezioni, la funzione deve essere *implementata*, ossia scritta dall'inizio alla fine, *prima* del suo primo utilizzo. Se la funzione `Quadrato` fossa stata scritta dopo il `main`, il compilatore avrebbe rilevato un errore;

2. l'argomento della funzione viene *modificato* nella funzione e ciò non provoca alterazioni della variabile definita nel `main`. Ciò dipende dal fatto che i valori sono passati alla funzione per valore e così facendo, sono stati copiati nella variabile locale della funzione;
3. le variabili del `main` hanno un identificatore diverso da quello dell'argomento della funzione. Non è sbagliato usare nella funzione chiamante ed in quella chiamata variabili con lo stesso nome, ma lo si sconsiglia vivamente, per evitare confusioni;
4. l'argomento della funzione può essere una costante, una variabile, una funzione o un'espressione.

Lo studente potrebbe, però, legittimamente chiedersi in quali occasioni conviene oppure si è costretti ad utilizzare il passaggio per valore. In due casi particolari:

- quando non si vuole che eventuali modifiche dell'argomento effettuate all'interno della funzione abbiano effetto anche esternamente alla funzione. In tal modo si "protegge" la variabile passata per valore;
- quando l'argomento passato è di piccole dimensioni, ovvero limitato a pochi byte. Se le dimensioni dovessero essere molto grandi (migliaia o milioni di byte) il passaggio per valore diventa inefficiente e *resource consuming*.

11.1.2 Il passaggio per riferimento

La sintassi relativa al passaggio per valore di un argomento ad una funzione è illustrato nella figura sottostante:

```
int StrLen(char *c)
{
    int len=0;

    while(*c++!='\0')
        len++;
    return len;
}
```


Figura 11.3: Passaggio per riferimento

Il passaggio per riferimento è caratterizzato dal fatto che all'argomento della funzione viene passato l'indirizzo della variabile originale, senza effettuare alcuna copia. Se, all'interno della funzione la variabile passata per riferimento viene modificata, viene modificato anche l'originale. Dal punto di vista sintattico la definizione di argomento è identica alla definizione di puntatore.

I casi in cui si deve/preferisce passare gli argomenti per riferimento sono grossolanamente i seguenti:

- quando si vuole che eventuali modifiche dell'argomento effettuate all'interno della funzione abbiano effetto anche esternamente alla funzione. In tal modo, le modifiche vengono operate direttamente sull'originale, ovvero sulla variabile passata alla funzione;
- quando l'argomento passato è di grandi dimensioni.

Si supponga il seguente esempio, nel quale la funzione `StrLen` ha il compito di calcolare la lunghezza di una stringa passata per riferimento:

Listing 11.2: Esempio di passaggio per riferimento

```
int StrLen(char *c)
{
   int len=0;   //Lunghezza della stringa

   //Si scandisce ciascun carattere della stringa dal primo
   //fino al terminatore, incrementando via via la lunghezza
   //della stringa. Quando si trova il terminatore si esce dal
   //ciclo.
   while(*c++)
      len++;    //Aggiorna la lunghezza della stringa

   return len;  //Restituisce la lunghezza della stringa
}

void main(void)
{
   char pippo[] = "Pippo";
   char topolino[] = "Topolino";
   int lgt;

   //Restituisce in lgt la lunghezza delle stringhe. Non viene
   //conteggiato il terminatore.
   lgt = StrLen(pippo);          //lgt=5
   lgt = StrLen(topolino);       //lgt=8
}
```

Si nota che la funzione `StrLen` definisce al proprio interno una variabile locale atta a contenere temporaneamente la lunghezza della stringa. Si ricordi, inoltre, che il nome del vettore è anche puntatore a se stesso.

11.1.3 Errori frequenti nell'uso delle funzioni

Gli errori che gli studenti compiono frequentemente a proposito delle funzioni sono svariati. La presente sezione cercherà di renderne puntuale resoconto, in modo da permettere al lettore di evitare di compiere gli errori elencati. Di alcuni si è in grado di dare giustificazione dell'errore compito, si è cioè in grado di immaginare "perché" venga effettivamente compiuto. Per quanto riguarda altri errori tipici, l'autore della presente non è proprio in grado di capirne la natura né il "perché".

1. **si utilizza lo stesso identificatore sia per la variabile passata alla funzione che per l'argomento definito nella funzione**. Non si tratta effettivamente di un errore sintattico, ma così facendo si alimenta un'inutile confusione. Un esempio di detto "errore" è il seguente:

Listing 11.3: Errore sugli identificatori

```
int Pippo(int pluto)
{
   pluto++;

   //Il corpo della funzione contiene altre istruzioni,
   //delle quali ci disinteressiamo perche' non importanti.

   return pluto;
}

void main(void)
{
   int pluto=3;

   pluto = Pippo(pluto); //pluto viene modificato?
}
```

Chiedersi se `pluto` viene modificato significa alimentare la confusione: quale `pluto`? Quello interno alla funzione `Pippo` o quello esterno ad essa? Solitamente lo studente assegna lo stesso identificatore perché non conosce bene l'argomento e gli sembra "più prudente" non prendere iniziative. Per togliersi definitivamente il dubbio, lo studente potrebbe tentare di assegnare lo stesso nome sia alla variabile passata che all'argomento della funzione nel codice 11.1. In tal caso noterà le inutili complicazioni nelle quali incorrerà;

2. **manca il *type matching* fra variabile passata e argomento della funzione**. Ciò succede frequentemente se l'argomento è un puntatore. Alcuni esempi sono i seguenti:

Listing 11.4: Errore di *type matching*

```
int StrLen(char *c)
{
   int len=0;   //Lunghezza della stringa

   //Si scandisce ciascun carattere della stringa dal
   //primo fino al terminatore, incrementando via via la
   //lunghezza della stringa.Quando si trova il terminatore
   //si esce dal ciclo.
   while(*c++)
      len++;    //Aggiorna la lunghezza della stringa

   return len;  //Restituisce la lunghezza della stringa
}

void main(void)
{
   char dotto[]="Dotto";
   int lgt;              //Lunghezza della stringa

   lgt=StrLen(&dotto);   //Errore (a)
   lgt=StrLen(dotto[0]); //Errore (b)
   lgt=StrLen(dotto[]);  //Errore (c)
```

```
   lgt=StrLen("Dotto");  //Errore (d)?
}
```

Esaminiamo i singoli errori:

(a) il passaggio `&dotto` è errato, perché si estrae l'indirizzo del puntatore e non dell'oggetto puntato. Per capire meglio quanto detto si faccia riferimento alla fig. 10.1 a pagina 278: a noi interessa il *contenuto* di `pippo`, ossia 0x2F88A600 e non il suo indirizzo, ossia 0x2C723E20. A noi interessa l'indirizzo dell'oggetto puntato, in questo caso una stringa, non del suo puntatore. La sintassi corretta sarebbe stata `lgt=StrLen(dotto);`. Si ponga attenzione: l'errore è diffusissimo;

(b) in questo caso si passa un *carattere* e non un *puntatore a carattere* come è richiesto dall'argomento della funzione. Anche in questo caso si tratta di un errore piuttosto frequente;

(c) la sintassi `dotto[]` non è accettata dal linguaggio C, anche se essa avrebbe un certo senso. Effettivamente si potrebbe obiettare che l'espressione `dotto[]` indica proprio un vettore. L'obiezione ha un certo valore, ma la sintassi del C non la accetta e tanto basta. Si noti che, invece, è correttissima la sintassi `int StrLen(char c[])`: essa è accettata nella definizione di argomento, ma non nel suo passaggio;

(d) alla luce di quanto detto la quarta espressione dovrebbe essere errata e, invece, è assolutamente corretta. Dal punto di vista concettuale l'espressione `"Dotto"` indica una *costante*, ma dal punto di vista sintattico l'espressione viene intesa come puntatore a carattere.

3. **definire variabili locali come argomenti della funzione**. Si tratta di un errore diffusissimo che chi scrive non ha mai capito. Si propone il seguente codice:

Listing 11.5: Errore nel numero degli argomenti

```
int StrLen(char *c, int len)
{
   len=0;        //Lunghezza della stringa

   //Si scandisce ciascun carattere della stringa dal primo
   //fino al terminatore incrementando via via la lunghezza
   //della stringa. Quando si trova il terminatore si esce
   //dal ciclo.
   while(*c++)
      len++;     //Aggiorna la lunghezza della stringa

   return len;  //Restituisce la lunghezza della stringa
}

void main(void)
{
   char dotto[]="Dotto";
   int lgt;                    //Lunghezza della stringa

   lgt=StrLen(dotto, lgt);     //Cervellotico!
}
```

La funzione `StrLen` abbisogna di una variabile temporanea per il calcolo della lunghezza della stringa. Tale variabile temporanea viene definita *fra gli argomenti della funzione* e non *fra le variabili locali*. Non si tratta di un errore sintattico, ma di un errore concettuale piuttosto grave. Si sottolinea con forza che **gli argomenti di una funzione rappresentano il suo collegamento col mondo esterno**. Se la funzione `RadiceQuadrata` necessita del solo argomento `radicando` *non si devono passare altri argomenti alla funzione*. Se c'è bisogno di una variabile locale nella funzione la si definisce come nel codice 11.4 a pagina 292 e non fra gli argomenti della funzione. Forse l'errore nasce dal dubbio che non sia possibile definire variabili nella funzione chiamata. Ciò è assurdo: tutte le funzioni permettono la definizione di variabili;

4. **dimenticare di ritornare alcun valore se previsto dalla funzione**. Non si tratta di un vero e proprio errore, ma di una dimenticanza, anche se frequente. Si veda il seguente esempio:

Listing 11.6: Errore di valore non ritornato

```
int StrLen(char *c)
{
   int len=0;   //Lunghezza della stringa

   //Si scandisce ciascun carattere della stringa dal primo
   //fino al terminatore incrementando via via la lunghezza
   //della stringa. Quando si trova il terminatore si esce
   //dal ciclo.
   while(*c++)
      len++;    //Aggiorna la lunghezza della stringa

   //Manca il return!
}

void main(void)
{
   char dotto[]="Dotto";
   int lgt;      //Lunghezza della stringa

   lgt=StrLen(dotto);
}
```

Nella funzione `StrLen` manca il `return`, per cui non viene ritornato il valore calcolato e relativo alla effettiva lunghezza della stringa. Si tratta di un errore sintattico che il compilatore evidenzia immediatamente. Si noti che il compilatore verifica che il `return` sia presente in tutti i punti di uscita. Si supponga che nella parte finale della funzione sia posta un'istruzione `if..else` e che si ponga il `return` solamente alla fine del corpo dell'`if` e non nell'`else`: il compilatore segnala errore in tal caso.

E' invece possibile la situazione opposta, in cui il valore ritornato dalla funzione chiamata non venga letto. Si possono ipotizzare dei casi in cui una determinata funzione esegue più di una azione e che non sia importante esaminare e trattare il valore da essa ritornato.
In tali casi si può non solo ignorare il valore di ritorno, ma proprio evitare di passarlo ad una variabile.

Si veda il seguente codice:

Listing 11.7: Valore ritornato e non letto

```
int ModStr(char *c)
{
   int len=0;   //Lunghezza della stringa

   //Si modifica la stringa cambiando la maiuscola in
   //minuscola.
   if((*c>='A')&&(*c<='Z'))
      *c -= 'A'-'a';

   //Si scandisce ciascun carattere della stringa dal primo
   //fino al terminatore incrementando via via la lunghezza
   //della stringa. Quando si trova il terminatore si esce
   //dal ciclo.
   while(*c++)
      len++;    //Aggiorna la lunghezza della stringa

   return len;
}

void main(void)
{
   char dotto[]="Dotto";
   int lgt;       //Lunghezza della stringa

   //Si noti che la funzione ModStr non assegna il valore
   //calcolato a nessuna variabile. Cio' e' permesso.
   ModStr(dotto); //"Dotto" viene modificato in "dotto"
}
```

La funzione `ModStr` ritorna la lunghezza della stringa, ma detto valore non viene "raccolto" da nessuna variabile;

5. **il numero degli argomenti della funzione chiamata non è uguale al numero di variabili passate alla funzione stessa**. Anche il presente è un errore piuttosto diffuso ed è probabilmente dovuto a distrazione o superficialità nel rispettare eventuali consegne. Un esempio di detto errore è il seguente:

Listing 11.8: Errore *matching* argomenti/variabili

```
int ModStr(char *c)
{
   int len=0;   //Lunghezza della stringa

   //Si modifica la stringa cambiando la maiuscola in
   //minuscola.
   if((*c>='A')&&(*c<='Z'))
      *c -= 'A'-'a';

   //Si scandisce ciascun carattere della stringa dal primo
   //fino al terminatore incrementando via via la lunghezza
   //della stringa. Quando si trova il terminatore si esce
   //dal ciclo.
```

```
   while(*c++)
      len++;    //Aggiorna la lunghezza della stringa

   return len;
}

void main(void)
{
   char dotto[]="Dotto";
   int lgt;       //Lunghezza della stringa

   //Il numero della variabili passate alla ModStr non e'
   //uguale al numero degli argomenti della ModStr.
   lgt=ModStr(dotto, lgt);
}
```

Nel presente caso alla funzione `ModStr` vengono passate due variabili, la prima per riferimento e la seconda per valore (ma perché?), mentre essa accetta solamente un argomento passato per riferimento;

6. **il nome della funzione viene storpiato**. L'errore più comune è quello di non rispettare le maiuscole/minuscole, come di seguito evidenziato:

Listing 11.9: Errore nel nome della funzione

```
void main(void)
{
   char dotto[]="Dotto";

   //La funzione in realta' si chiama ModStr
   //e non modstr
   modstr(dotto);
}
```

Naturalmente non c'è limite alla fantasia di chi produce errori. Quelli presentati sono quelli che più frequentemente si incontrano.

11.1.4 Un esempio d'uso delle funzioni

L'esercizio proposto è un po' diverso dai soliti. Viene presentato, insieme al problema da risolvere, anche un `main` già pronto, ma senza le rispettive funzioni, che sono da scrivere in modo che si adattino al predetto `main`.

Esercizio - ◊◊◊ Calcolo di serie

Della già nota serie[1], indicata al punto 11.1, si chiede:

$$1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + ... + \frac{1}{2^{n-1}} \tag{11.1}$$

1. il diagramma di flusso e la scrittura della funzione `DigitInt` che permetta all'utente di digitare un numero n compreso fra 5 e 20;
2. il diagramma di flusso e la scrittura della funzione `Potenza`, avente due parametri, b ed e, che permetta il calcolo di b^e, con $b \in N, b \mid b > 0$ e $e \in N, e \mid e \geq 0$;

[1] Vedi esercizio proposto nella sezione 8.3

3. il diagramma di flusso e la scrittura della funzione `Serie` che utilizzi la funzione `Potenza` per calcolare la serie di *n* elementi;
4. la sola scrittura di una funzione `Stampa` che visualizzi il risultato, come indicato nel sottostante codice.

Si fornisce il `main` che richiama le suddette funzioni, per cui queste devono adattarsi alle espressioni chiamanti.

Listing 11.10: Calcolo di serie

```
int main()
{
   const int kLimInf = 5;       //Limite inferiore
   const int kLimSup = 20;      //Limite superiore

   int n;       //Numero compreso fra kLimInf e kLimSup
   double s;    //Risultato fornito dalla serie

   //Chiede all'utente di inserire un numero compreso fra
   //kLimInf e kLimSup
   n = DigitInt(kLimInf, kLimSup);

   //Calcola la serie di n elementi e la restituisce in s
   s = Serie(n);

   //Stampa il risultato nella forma:
   //''La serie di ... elementi vale ...'' con il valore
   //di n e s al posto dei puntini
   Stampa (n, s);

   return 0;
}
```

Soluzione

Il diagramma della funzione `Immissione` è già stato sostanzialmente proposto in fig. 8.10 a pagina 238 per cui si ritiene sufficiente riproporlo in fig. 11.4 nella pagina successiva, senza l'aggiunta di commenti. La scrittura della relativa funzione assume la seguente forma:

Listing 11.11: Serie: Immissione

```
int DigitInt(int liminf, int limsup)
{
   int m;

   //Ciclo di inserimento del numero mediante ciclo do..while
   do
   {
      //Frase di cortesia. Viene scritta su due righe per
      //motivi grafici imputabili alla presente pagina.
      printf("Digitare␣un␣numero␣compreso␣fra␣%d", liminf);
      printf("e␣%d", limsup);
      scanf("%d", &m);
   }while(m<liminf||m>limsup);
   return m;    //Ritorna il numero
}
```

Non ci concentreremo più di tanto sul corpo della funzione, dando per scontato l'algoritmo vero e proprio. In questo momento ci interessa di più il titolo della funzione.

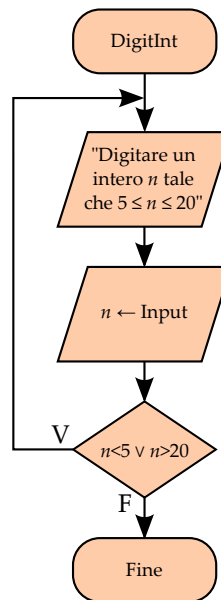



Figura 11.4: Serie: Funzione DigitInt

Il nome della funzione identifica in maniera mnemonica e grossolana il compito che essa svolge. Ciò è importante quando si scrivono parecchie migliaia di righe di codice, perché permette di non dover documentarsi continuamente su ciò che la funzione fa.

La `DigitInt` ci suggerisce, attraverso il proprio nome, che essa permetta la digitazione di un numero intero. Il tipo di ritorno conferma questa ipotesi. Leggendo gli argomenti della funzione si dovrebbe riuscire a capire anche quale sia la caratteristica del numero di digitare. Si tratta di due costanti intere denominate `kLimInf` e `kLimSup`. Dal tipo e dal nome dei due argomenti si può ipotizzare che essi rappresentino un limite intero inferiore ed un limite intero superiore caratterizzanti il numero da digitare.

E' molto importante sottolineare che i due argomenti passati alla funzione sono e devono essere *l'unico tramite con il mondo esterno*. La funzione non è autorizzata a sapere nulla del mondo a sé esterno se non quanto passato attraverso la lista degli argomenti. Pertanto, questi ultimi, devono essere scelti con cura, senza ridondanze né mancanze, alle quali supplire poi, erroneamente, mediante goffi accessi a variabili o costanti globali. Inoltre, se possibile ed efficiente, gli argomenti dovrebbero essere passati per valore, orientandosi al passaggio per riferimento solo nei casi di grandi dimensioni o quando si vuole espressamente che la funzione modifichi direttamente la variabile puntata.

Se quanto fin qui detto è chiaro, si può procedere nella definizione della funzione `Potenza`. Le consegne dicono che detta funzione deve avere due parametri, b ed e, e permettere il calcolo di b^e, con $b \in N$, $b \mid b > 0$ e $e \in N$, $e \mid e \geq 0$.

Non si tratta di un problema di straordinaria complessità: è sufficiente calcolare il risultato parziale di ripetute moltiplicazioni, dopo averlo inizializzato ad 1. Le "ripetute moltiplicazioni" suggeriscono la presenza di un ciclo. Si tratta quindi di stabilire di che ciclo si tratti. Siccome il numero delle moltiplicazioni è noto a priori e vale `e`, si tratterà di un ciclo `for`. Un possibile diagramma di flusso potrebbe essere il seguente:

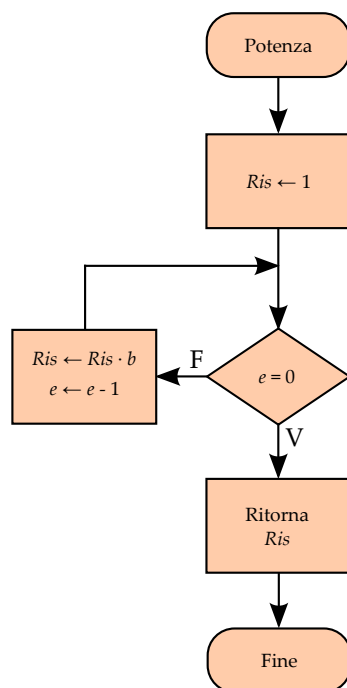



Figura 11.5: Serie: Funzione Potenza

Inizialmente il risultato parziale `Ris` viene inizializzato a 1, in modo da, se si dovesse avere $e = 0$, non entrare nel corpo del ciclo ed uscire con `Ris=0`, che rappresenta il risultato corretto in presenza di b^e. Se si dovesse avere $e > 0$ si moltiplicherà il risultato parziale per b tante volte quante vale e.

Dal diagramma di fig. 11.5 è possibile passare al codice in linguaggio C:

Listing 11.12: Funzione Potenza

```
//Calcolo dell'elevamento a potenza b^e, con b appartenente a N,
//tale che b > 0 ed e appartenente a N, tale che e >= 0. Si noti
//che la funzione ritorna volutamente un intero in modo da porre
//in evidenza che si deve categoricamente evitare la divisione
//intera per ciascun termine.
```

```
int Potenza(int b, int e)
{
   int ris=1;   //Risultato dell'elevamento a potenza
   int i;       //Indice del ciclo for

   //Anche in questo caso si noti la scelta del ciclo.
   //Se e=0, non si entra mai nel corpo del for ed il
   //risultato e' 1.
   for(int i=0; i<e; i++)
      ris *= b;

   return ris;
}
```

Anche in questo caso si cercherà di mantenere il *focus* sul titolo della funzione, che rappresenta il tema centrale del presente capitolo.

Il nome della funzione esplica piuttosto chiaramente quale sia il compito al quale la funzione assolve: calcolare un elevamento a potenza. La base e l'esponente sono indicati fra i parametri con `b` ed `e`. Il valore ritornato è di tipo intero, dato che base ed esponente sono interi.

La funzione utilizza due variabili locali: `ris` e `i`, rispettivamente per il risultato parziale e per l'indice del ciclo `for`. Si noti che dette variabili *non sono definite fra i parametri della funzione*. Definirle fra gli argomenti rappresenterebbe un grave errore concettuale. Una funzione `Potenza` necessita solamente della base e dell'esponente per calcolarne l'elevamento a potenza, quindi è corretto che nella lista degli argomenti compaiano solo e solamente detti due parametri.

E' importante che lo studente maturi la capacità di individuare gli argomenti strettamente necessari da passare ad una determinata funzione. Nel presente esempio ciò è molto semplice, dato che il `main` evidenzia già chiaramente nel testo quali debbano essere gli argomenti da passare alla funzione, ma non è sempre così. Di solito è compito del programmatore prendere tali decisioni. Non è quindi prematuro esortare il futuro programmatore a sviluppare detta sensibilità.

Ora si può sviluppare l'algoritmo della serie vera e propria. Esso è già stato proposto in maniera analoga nella sezione 8.3, per cui si può supporre che il lettore non sia preso totalmente alla sprovvista. Il calcolo di una serie coinvolge sempre un'iterazione. Se si conosce il numero di termini da elaborare si dovrà utilizzare un'iterazione a numero di cicli definiti a priori.

Nel presente caso il numero di termini è indicato dalla variabile `n` che viene passata come argomento alla funzione `Serie`, per cui si utilizzerà un ciclo `for`.

Nel corpo del ciclo si dovrà calcolare prima il denominatore del singolo termine e successivamente il termine vero e proprio. Il calcolo del denominatore coinvolge la funzione `Potenza` appena elaborata. Essa dovrà calcolare il valore 2^i dove i è l'indice del ciclo `for`, che assume i valori che vanno da 0 a $n-1$.

Durante il calcolo del singolo termine si dovrà porre attenzione che la divisione non sia di tipo intero. Si è già visto che ciò si può ottenere quando o il numeratore o il denominatore è in virgola mobile. Il modo più semplice per ottenere detta condizione è porre il numeratore al valore 1.0.

Alla luce di quanto fin qui detto, un possibile diagramma di flusso della funzione `Serie` potrebbe essere quello rappresentato nella figura 11.6.

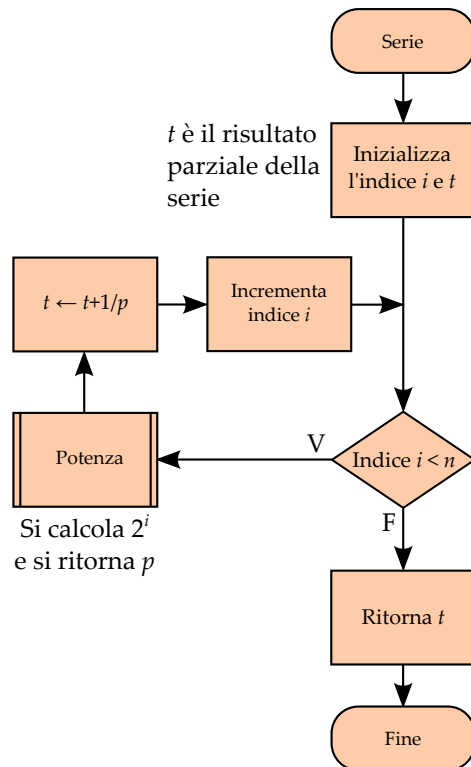



Figura 11.6: Serie: Funzione Serie

Dato che si suppone che il diagramma di flusso presentato non rappresenti un'eccessiva difficoltà, si presenta il codice che implementa detto diagramma:

Listing 11.13: Funzione Serie

```
//Calcolo della serie numerica indicata nelle specifiche con
//un numero di termini pari a quello indicato a parametro.
//All'aumentare del numero di termini la serie numerica si
//avvicina sempre piu' a 2.

double Serie(int t)
{
   double ris = 0.0;    // Risultato della serie
   int i;               //Indice del ciclo for

   //Anche in questo caso si noti la scelta del ciclo.Il numero
   //delle iterazioni e' noto a priori. Si noti anche che nel
   //corpo del ciclo il numeratore della frazione non e' 1 ma
   //1.0. Se cosi' non fosse, la frazione sarebbe calcolata
   //come divisione fra interi e quindi l'operatore verrebbe
   //considerato intero con esiti catastrofici: ciascun termine
   //dal secondo in poi varrebbe sempre 0.
```

```
   for(i=0; i<t; i++)
      ris += 1.0/Potenza(2, i);

   return ris;
}
```

Come si nota le istruzioni sono ridotte al minimo. Il codice poteva essere ulteriormente compresso definendo ed inizializzando le variabili `ris` ed `i` nell'istruzione `for`. Si preferisce, però, rendere più leggibile il codice anche a prezzo di qualche riga in più.

Il codice della funzione `Stampa` è piuttosto banale e lo si propone di seguito:

Listing 11.14: Funzione Stampa

```
//Stampa il risultato nella forma:
//"La serie di ... elementi vale ..." con t e ris al posto dei
//puntini di sospensione.

void Stampa(int t, double ris)
{
   printf("La␣serie␣di␣%d␣elementi␣vale␣%f", t, ris);
}
```

L'unica particolarità degna di nota è data dalla mancanza di valore ritornato. In tal caso il tipo di ritorno della funzione deve essere `void`.

11.2 Funzioni e vettori

Si è già visto parzialmente come si possono passare i vettori alle funzioni e come si possa accedere ai relativi elementi. In realtà l'argomento non è stato sviluppato esaustivamente, ed è giunto il momento di farlo.

Fino ad ora si è affrontato l'argomento con le seguenti limitazioni:

1. si sono sempre passate delle stringhe alla funzione;
2. si sono sempre passati dei vettori monodimensionali alla funzione;
3. nel corpo della funzione l'accesso al singolo elemento è sempre avvenuto mediante puntatore.

Si tratta, quindi, di illustrare 1) come passare alla funzione vettori generici, che non contengano quindi solo stringhe; 2) come si possano passare dei vettori multidimensionali alla funzione; 3) se e come l'accesso al singolo elemento possa avvenire senza l'uso di puntatori.

11.2.1 Funzioni e vettori generici

Negli esempi fin qui visti si sono passati alle funzioni sempre vettori contenenti stringhe. Quest'ultima contiene un'informazione implicita sulla sua effettiva lunghezza: mediante il terminatore è possibile stabilire dove la stringa finisce e quindi calcolare anche la sua lunghezza. Il vettore monodimensionale generico (quindi diverso dalla stringa), invece, non contiene alcuna indicazione sulla propria lunghezza. Se la funzione non è autorizzata a conoscere la lunghezza

del vettore, tale informazione deve essere passata come parametro insieme al vettore stesso, come nel seguente esempio:

Esercizio - ◊◊◊ Riempimento di vettore

Si chiede la definizione del diagramma di flusso e la scrittura del programma, adeguatamente suddiviso in funzioni, che:

1. chieda all'utente di definire un intero positivo n minore o uguale a 10;
2. esorti l'utente a digitare n numeri interi;
3. calcoli la media dei numeri digitati;
4. stampi i numeri digitati e la loro media aritmetica.

Soluzione

Si fa notare che la presenza del vettore è indispensabile e non opzionale o consigliata per motivi didattici. L'ultima richiesta riguarda la stampa della media aritmetica *e dei numeri digitati*, il che significa che detti numeri devono essere memorizzati da qualche parte. Se poi qualche studente burlone volesse far uso di 10 variabili, si potrebbe modificare l'esercizio e chiedere l'immissione di **100** numeri anziché 10.

Dato che viene espressamente chiesto, si inizia con il suddividere il problema in sottoproblemi, ovvero funzioni. Il fatto che l'esercizio lo chieda esplicitamente ha funzioni puramente didattiche: in realtà, la suddivisione in funzioni di un programma viene effettuata per *default* dal programmatore, anche non esperto. Si tratta di una applicazione più o meno conscia del metodo cartesiano che, nell'analisi, prevede proprio la *suddivisione* in sotto-problemi, isolatamente più semplici, di un problema dato.

Le funzioni potrebbero essere in minor numero rispetto alle richieste:

- **funzione DefIntNum** che permetta la definizione del numero positivo minore o uguale a 10;
- **funzione PutInVect** che memorizzi gli n numeri nel vettore e calcoli la media aritmetica dei numeri digitati;
- **funzione Print** che stampi i numeri digitati e la media aritmetica.

Contestualmente alla suddivisione in funzioni del programma si potrebbero definire gli argomenti necessari a ciascuna funzione e se debbano o meno restituire un valore:

DefIntNum - come già visto in altri esercizi, sarebbe bene che gli argomenti fossero 2: un limite inferiore ed un limite superiore. Il primo limite si rende necessario perché il numero deve essere *positivo* piuttosto che *non negativo*. E' bene che entrambi gli argomenti vengano passato per valore. La funzione dovrà restituire un intero rappresentante il numero digitato.

PutInVect - anche in questo caso si rendono necessari due argomenti: il vettore, passato per riferimento, e il numero effettivo di numeri da porre nel vettore, passato per valore. Tale numero non è detto che coincida con la dimensione del vettore, ma potrebbe essere anche inferiore. La funzione dovrà ritornare un numero in virgola mobile rappresentante la media aritmetica calcolata.

Print - saranno necessari 3 argomenti: il vettore, passato per riferimento; il numero effettivo di numeri contenuti nel vettore, passato per valore; la media aritmetica, passata per valore. Non è necessario che la funzioni ritorni alcun valore.

La prima parte del problema è già stata affrontata in precedenti esercizi, per cui si ritiene inutile commentare ulteriormente il diagramma di flusso, che potrebbe essere il seguente:

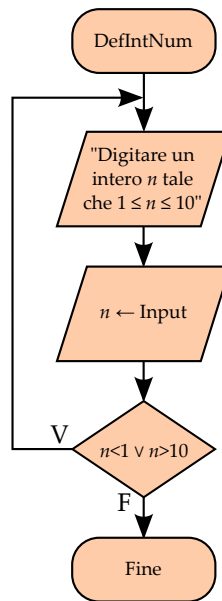



Figura 11.7: Vettore: Funzione DefIntNum

Analogo discorso vale per la funzione `DefIntNum` che assomiglia molto alla funzione `DigitInt` dell'esercizio precedennte:

Listing 11.15: Vettore: Funzione DefIntNum

```
int DefIntNum(int liminf, int limsup)
{
   int m;

   //Ciclo di inserimento del numero richiesto.
   //Si tratta di un ciclo do..while.
   do
   {
      //Frase di cortesia. Viene scritta su due righe per
      //motivi grafici imputabili alla presente pagina.
      printf("Digitare␣un␣numero␣compreso␣fra␣%d", liminf);
      printf("␣e␣%d:␣", limsup);
      scanf("%d", &m);
   }while(m<liminf||m>limsup);

   return m;     //Ritorna il numero
}
```

Si trova utile effettuare una breve riflessione su quanto fatto. Se si confrontano le due funzioni `DefIntNum` e `DigitInt` dell'esercizio precedente, si noterà che sono identiche. Ciò significa che quando si sono pensate le due funzioni esse sono state scritte *cercando di generalizzarle il più possibile*. Così facendo

si aumenta la probabilità di creare una funzione *riutilizzabile*, ossia una funzione di libreria. D'ora in poi si darà per scontata la funzione di immissione di numero intero.

L'immissione di un numero noto di valori interi implica l'uso di un ciclo `for`. Il diagramma di flusso relativo ad una simile funzione potrebbe essere il seguente:

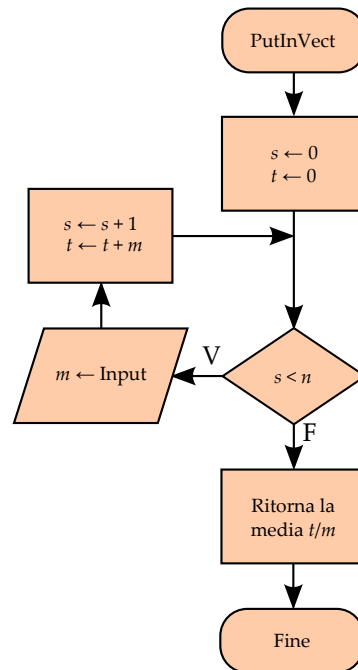



Figura 11.8: Vettore: Funzione PutInVect

Si tratta del classico ciclo `for` con indice `i`, che esegue `n` volte il corpo del ciclo. Il codice relativo ad una siffatta funzione potrebbe essere il seguente:

Listing 11.16: Vettore: Funzione PutInVect

```
double PutInVect(int *vet, int len)
{
   double t=0.0; //Somma dei numeri digitati
   int i;

   //Frase di cortesia. L'utente viene invitato
   //a digitare len numeri interi.
   printf("Digitare␣%d␣numeri␣interi:\n", len);

   //Ciclo di inserimento dei valori nel vettore
   //mediante ciclo for. Si noti la sintassi
   //della scanf.
   for(i=0; i<len; i++)
   {
      scanf("%d", vet+i);
```

```
      t += *(vet+i);     //Aggiorna la somma
   }

   //Calcola e ritorna la media
   return t/len;
}
```

Una parte degna di commento è rappresentata dalla funzione `scanf`. Il secondo argomento della funzione richiede il puntatore all'area di memoria in cui memorizzare il numero digitato. Tale area è individuata con la notazione `vet+i`. Essa utilizza l'indice del ciclo `for` e l'aritmetica dei puntatori per individuare l'elemento del vettore in cui memorizzare il numero digitato: quando l'indice vale 0, `vet+i` punta il primo elemento del vettore; quando l'indice vale 1, punta il secondo e così via.

All'interno del corpo del ciclo viene anche aggiornata la somma dei numeri digitati, in modo da poter calcolare la media alla fine del ciclo. In questo caso la sintassi è leggermente diversa se si vuole sommare a `t` l'elemento puntato, ossia `*(vet+i)`. E' importante che si riconosca senza incertezze il significato delle due sintassi: `vet+i` significa puntatore all'i_{esimo} elemento del vettore; `*(vet+i)` significa i_{esimo} elemento del vettore.

Ma la vera differenza con le altre funzioni che hanno vettori in argomento è data dalla presenza della lunghezza effettiva dello stesso, rappresentata dal parametro `len`. Dobbiamo, infatti, distinguere fra la dimensione del vettore e l'effettiva area occupata. Nel presente caso la dimensione del vettore è pari a 10 interi, ossia 40 byte. L'effettiva occupazione dello stesso dipende però dal numero digitato inzialmente, che abbiamo abbinato alla variabile n. Se l'utente ha digitato il valore 7, l'effettiva occupazione del vettore sarà pari a 7 interi, ovvero 28 byte. Il parametro `len` è quindi fondamentale.

L'ultima funzione ha il compito di stampare i numeri digitati e la media aritmetica calcolata. Anche in questo caso l'attenzione va posta sul numero e significato degli argomenti della funzione, piuttosto che sul diagramma di flusso (vedi fig. 11.9 a fronte).

A dire il vero basterebbero due soli parametri per stampare quanto dovuto: il vettore e la sua lunghezza. Il terzo parametro, ossia la media aritmetica, serve solo a non rifare dei conti già fatti, ovvero per ottimizzare il compito. Quindi quando si dice che alla funzione vanno passati "solo gli argomenti strettamente necessari", lo si deve fare *cum grano salis*. Nel presente caso passare la media già calcolata è indubbiamente comodo.

Per quanto riguarda il valore di ritorno, invece, ci si può comportare come per la funzione `Stampa` dell'esercizio precedente, non ritornando alcun valore.

Un possibile esempio di come si potrebbe sviluppare la funzione `Print` è il seguente:

Listing 11.17: Funzione Print

```
//Stampa l'elenco dei valori digitati e la media.

void Print(int *vet, int len, double a)
{
   int i;
```

```
        //I numeri digitati sono stampati mediante un
        //ciclo for.
        printf("La␣media␣aritmetica␣dei␣valori\n");
        for(i=0; i<len; i++)
           printf("%d␣␣", *(vet+i)); //Si notino gli spazi
                                    //dopo %d
        printf("e'␣%f", a);
     }
```

Come per la funzione `PutInVect`, anche la presente esegue l'accesso ai singoli elementi mediante la notazione `*(vet+i)`, con `i` che varia fra 0 e `len-1`.

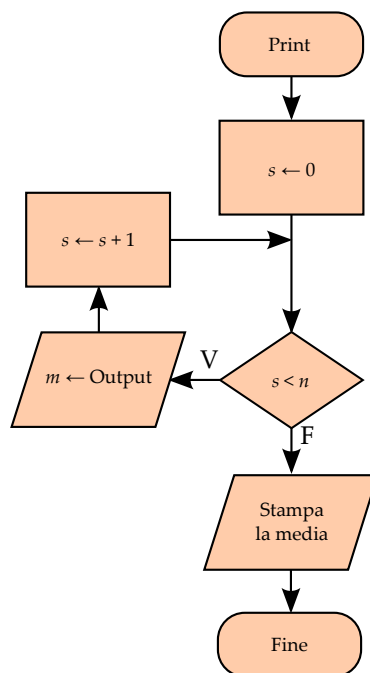



Figura 11.9: Vettore: Funzione Print

E' ora possibile concludere l'esercizio scrivendo il `main` avente il compito di lanciare le funzioni appena descritte:

Listing 11.18: Riempimento di vettore

```
int main()
{
   const int kLimInf = 1;        //Limite inferiore
   const int kLimSup = 10;       //Limite superiore

   int vettore[kLimSup]; //Vettore contenente i valori digitati
   int n;                //Numero compreso fra kLimInf e kLimSup
   double avg;           //Media dei valori digitati

   //Chiede all'utente di inserire un numero compreso fra
```

```
    //kLimInf e kLimSup
    n = DefIntNum(kLimInf, kLimSup);

    //Inserisce n valori nel vettore e restituisce la media
    avg = PutInVect(vettore, n);

    //Stampa i numeri digitati e la media
    Print(vettore, n, avg);

    return 0;
}
```

11.2.2 Funzioni e vettori multidimensionali

La filosofia che ispira le presenti pagine è quella di presentare al lettore, se possibile, un problema alla volta. Quanto verrà proposto nella presente sezione e relativo ai vettori multidimensionali, in realtà si adatta perfettamente anche ai vettori monodimensionali.

Fino ad ora si è sempre ipotizzato, correttamente, che la funzione non conosca la dimensione dei vettori passati in argomento. E' però altrettanto corretto ipotizzare che determinati vettori siano di lunghezza nota e costante. Si supponga, ad esempio, un vettore bidimensionale di interi che funga da controllo del traffico settimanale ad un casello autostradale: la prima dimensione potrebbe rappresentare i 7 giorni della settima e la seconda le 24 ore della giornata. Un tal vettore potrebbe essere definito nel seguente modo: `int traffico[7][24];`. Non ha alcun senso ipotizzare che cambi il numero dei giorni che compongono una settimana o il numero di ore in un giorno.

Questa osservazione ci suggerisce la possibilità che si possa dichiarare in maniera alternativa un siffatto vettore e ci introduce al seguente esempio di codice:

Listing 11.19: Modi alternativi di passaggio per riferimento

```
void Acquisizione(int *traffico);
void Acquisizione(int traffico[][24]);
void Acquisizione(int traffico[7][24]);
```

Il primo prototipo di funzione è quello classico e già noto. L'informazione che l'argomento veicola è semplice: si tratta di un puntatore ad un intero. Il puntatore potrebbe puntare un semplice intero, un vettore monodimensionale, un vettore multidimensionale e così via. E' compito della funzione gestire l'accesso alla tabella bidimensionale attraverso detto puntatore. Non è detto che si tratti di una complicazione o di una gestione inefficiente del vettore. Certamente, però, chi legge il codice non viene informato esplicitamente, attraverso il prototipo della funzione, che il puntatore punta un vettore bidimensionale, né tanto meno della lunghezza delle singole dimensioni.

Il secondo prototipo contiene qualche informazione in più. Informa che l'argomento è un vettore bidimensionale e che la seconda dimensione è formata da 24 elementi. Non si evince dal prototipo la lunghezza della prima dimensione. Si tratta comunque di un passo avanti dal punto di vista informativo rispetto al primo prototipo. Non confonda la notazione: si tratta sempre di

un passaggio per riferimento, quindi qualsiasi modifica effettuata sul vettore nella funzione, modifica il vettore originale.

Il terzo prototipo è il più chiaro in assoluto. Porta indicazione del fatto che si tratta di un vettore bidimensionale come pure delle lunghezze delle due dimensioni.

Le tre notazioni sono assolutamente identiche dal punto di vista funzionale, ma sono portatrici di tre contenuti informativi assolutamente differenti. Se le due dimensioni sono note alla funzione è sicuramente preferibile la terza notazione. Se solo una dimensione fosse nota alla funzione sarebbe corretto usare la seconda notazione. Se l'argomento è totalmente generico si utilizzerà la prima notazione. In quest'ultimo caso si renderanno necessari ulteriori due argomenti che portino l'informazione delle due dimensioni.

Esercizio - ◊◊◊ Tabellina di moltiplicazione

Si chiede la scrittura della funzione che provveda a "riempire" un vettore bidimensionale rappresentante le tabelline aritmetiche della moltiplicazione e della funzione che provveda poi a stamparla.

Soluzione

Dal punto di vista strutturale detta tabellina è un vettore bidimensionale di $10x10$ definito nel seguente modo: `char tabellina[10][10];`. Si tratta di scrivere 10 righe da 10 colonne ciascuna. A tal fine potrebbe essere utile ricorrere alla seguente definizione:

Listing 11.20: Definizione di kVert e kOriz

```
//Le sottostanti definizioni devono essere
//scritte prima delle funzioni.
#define kVert 10
#define kOriz 10
```

Sono quindi note le due dimensioni del vettore, per cui è immediato pensare a due cicli `for` annidati come nel codice 11.21:

Listing 11.21: Tabellina di moltiplicazione

```
void Tab(int vet[kOriz][kVert])
{
   char i, j;
   int *v;

   v = &vet[0][0];
   for(i=0; i<kOriz; i++)
      for(j=0; j<kVert; j++)
         *(v+i*kOriz+j)=(i+1)*(j+1);
}
```

La parte più enigmatica è il corpo del secondo `for` dove il prodotto degli indici (aumentati di 1) viene memorizzato in `*(vert+i*kOriz+j)`. L'espressione è assolutamente corretta, dato che `vet` punta ad inizio tabella che, si ricorda, è formata da kOriz vettori consecutivi (rappresentanti le kOriz righe), ciascuno dei quali contiene a sua volta kVert elementi. I due indici sono aumentati di 1 affinché il prodotto avvenga fra fattori compresi fra 1 e 10 e non fra 0 e 9.

Del tutto analoga è la funzione di stampa:

Listing 11.22: Stampa della tabellina

```
void PrintTab(int vet[kOriz][kVert])
{
   char i, j;
   int *v;

   v = &vet[0][0];
   for(i=0; i<kOriz; i++)
   {
      //Stampa la singola riga
      for(j=0; j<kVert; j++)
         printf("%3d␣", *(v+i*kOriz+j));

      //Arrivato a fine riga, va a capo
      printf("\n");
   }
}
```

Anche in questo caso il corpo del secondo `for` contiene la stessa espressione. Tutti i valori dell'i_{esimo} "vettore" vengono stampati sulla stessa riga ed alla fine si stampa un *carriage return* per andare a capo.

Un possibile `main` atto a lanciare le due funzioni appena descritte potrebbe essere il seguente:

Listing 11.23: Funzione main del programma tabellina

```
int main(void)
{
   char tabellina[kOriz][kVert];

   //Crea la tabellina
   Tab(tabellina);

   //Stampa la tabellina
   PrintTab(tabellina);

   return 0;
}
```

Si noti come il passaggio per riferimento dell'argomento avvenga sempre allo stesso modo.

11.2.3 Accesso al singolo elemento del vettore nelle funzioni

Nell'esercizio della sezione precedente si è visto come l'uso dei puntatori in presenza di vettori multidimensionali non sia molto intuitivo. In realtà l'accesso al singolo elemento del vettore può avvenire anche in altro modo. Nel codice sottostante, ad esempio, si ripropone la creazione della tabellina dell'esercizio precedente, senza l'uso della sintassi classica del puntatore:

Listing 11.24: Tabellina di moltiplicazione senza puntatore

```
void Tab(int vet[kOriz][kVert])
{
   char i, j;
```

```
        for(i=0; i<Oriz; i++)
           for(j=0; j<Vert; j++)
              vet[i][j]=(i+1)*(j+1);
}
```

Il corpo del secondo `for` non utilizza più la sintassi classica dei puntatori, ma quella meno complessa relativa ai vettori. L'accesso al vettore avviene sempre per riferimento, ma la sintassi è un po' più semplice.

11.3 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 11.

Capitolo 12

Le strutture

Nei precedenti capitoli i dati di *input* avevano tutti due caratteristiche in comune:

- erano omogenei;
- ciascun dato singolo rappresentava una porzione piccola ma indipendente del dato complessivo.

Si supponga, ad esempio un vettore contenente una parte della successione di Fibonacci, come illustrato in fig. 12.1:

0	1	1	2	3	5	8	13	21	34	55	89

Figura 12.1: Successione di Fibonacci

Se consideriamo il sottoinsieme 2, 3, 5, 8, 13, 21, esso rappresenta una porzione dell'informazione contenuta nel vettore. Se invece consideriamo la sequenza 5, 8, 13, la porzione dell'informazione è minore, ma mantiene una sua validità, si è, ad esempio, ancora in grado di riconoscere una piccola parte della sequenza di Fibonacci. Se si considera solamente il numero 8, la quantità di informazione rappresentata è veramente minima, ma è ancora possibile dire se l'elemento osservato appartiene o meno alla sequenza di Fibonacci.

Il lettore attento potrebbe chiedersi se i termini "dato" e "informazione" sono stati usati a caso o no.

12.1 Dati e informazioni

Nell'esempio citato è lecito "confondere" i termini "dato" e "informazione". Ma non è sempre così. Solitamente, infatti, non è lecito confondere i due termini, perché si riferiscono a concetti diversi fra loro. Il seguente esempio cercherà di evidenziarne le differenze.

Si suppongano i seguenti due dati:

- 0039 348 9876543;
- "Mario Rossi";

I suddetti dati identificano chiaramente una voce di una rubrica telefonica. Tali dati forniscono *insieme* un'informazione, che perde di significato nel momento in cui tali dati vengono separati. Se si suppone di memorizzare senza alcuna logica tutti i nomi in un vettore e tutti i numeri di telefono in un altro vettore, le informazioni contenute nella rubrica andrebbero perse: i dati relativi ai numeri sarebbero inservibili se separati dai rispettivi nomi.

Il linguaggio C prevede delle strutture dati utili a mantenere l'unità dell'informazione. Tali strutture si chiamano `struct`, dalla contrazione del termine inglese *structure*.

12.2 Le *struct*

Le strutture utilizzano la parola chiave `struct` e sono delle collezioni di dati che possono essere eterogenei fra loro dal punto di vista della tipologia, ma che assumono il loro pieno senso solo se mantenuti insieme. Un esempio di struttura dati potrebbe essere la seguente:

Figura 12.2: Esempio di struttura

La prima cosa che si nota è l'eterogeneità dei *campi* della struttura.[1] Tale caratteristica marca una netta differenza con il vettore che, per sua natura, contiene solo variabili dello stesso tipo. La struttura, invece, può contenere campi appartenenti a tipi differenti. Questa particolarità le permette di mantenere uniti tutti i dati relativi ad una determinata informazione.

[1]Parlando di strutture il termine *campo* è sinonimo di *variabile*.

Anche l'accesso ai singoli campi è diverso rispetto ai vettori: non c'è alcun indice che punti un determinato elemento del vettore, ma un meccanismo alquanto diverso. Prima, però di introdurre tale meccanismo è bene introdurre la modalità di dichiarazione della struttura.

12.2.1 La dichiarazione della struttura

In introduzione di capitolo si è parlato della sintassi della struttura e si è fornito un esempio esplicativo di *definizione* di struttura. La presente sezione parlerà, invece, della *dichiarazione* della stessa. La differenza, spesso trascurata dagli studenti, è stata abbondantemente trattata nella sezione 2.1. Lo studente che non ha ben chiara la differenza fra i suddetti due concetti farebbe, quindi, bene a rivedersi la corrispondente teoria.

In fig. 12.2 nella pagina precedente è evidenziata la definizione della struttura `Anagrafica`. In tal modo è stato reso noto al compilatore sia l'identificatore associato al *tipo della struttura* che le sue caratteristiche logiche (operatori permessi dominio di ciascun campo, ecc.). Per poter utilizzare la struttura `Anagrafica` è però necessario creare un contenitore per ciascuno dei valori che la struttura può ospitare. A tal fine è necessario creare una variabile e riservare ad essa dello spazio adeguato in memoria. Tale operazione avviene attraverso una normale operazione di dichiarazione:

Figura 12.3: Dichiarazione di struttura

Non vi è alcuna differenza con le dichiarazioni già viste nella sezione 2.1: a sinistra compare il tipo della variabile e a destra il suo identificatore. La seconda dichiarazione è relativa ad un vettore di anagrafiche.

12.2.2 La *dot notation*

Il meccanismo che permette l'accesso ai singoli dati componenti l'informazione è denominato *dot notation* ed è piuttosto semplice. In fig. 12.4 è evidenziato un

Figura 12.4: Esempio di *dot notation*

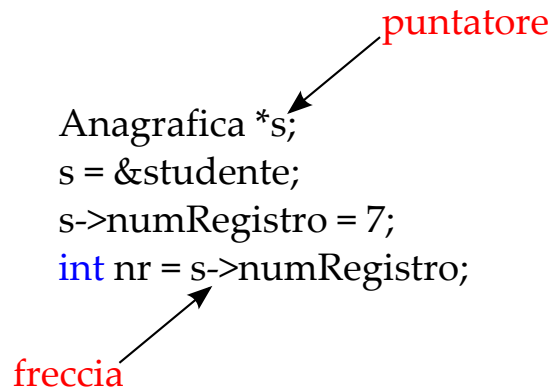
esempio di tale notazione sintattica: la prima riga rappresenta una scrittura del campo `numRegistro` della struttura `Anagrafica`, mentre la seconda riga rappresenta una lettura di detto campo. Il punto (*dot* finge da separatore fra l'identificatore della variabile e quello del campo della struttura.

12.2.3 Un esempio un po' più articolato

Il seguente è un esempio un po' più articolato di uso delle strutture. Si suppone di far uso della solita struttura `Anagrafica` e di voler inizializzare il vettore `classe` dichiarato nell'esempio di fig. 12.4.

Listing 12.1: Un esempio di uso di strutture

```
Anagrafica classe[30];
...
// I primi tre campi vanno inizializzati
// uno ad uno.
classe[0].cognome = "Abbagnale";
classe[0].nome = "Mario";
classe[0].dataNascita = 19981021;

classe[1].cognome = "Barberini";
classe[1].nome = "Giorgio";
classe[1].dataNascita = 19970506;

classe[2].cognome = "Brambilla";
classe[2].nome = "Marta";
classe[2].dataNascita = 19980909;
...
classe[29].cognome = "Zuzzi";
classe[29].nome = "Gianni";
classe[29].dataNascita = 19970425;

// Inizializza gli ultimi due campi
for(int i=0; i<30; i++)
{
   classe[i].classe = "3^TEL/B";
   classe[i].numRegistro = i+1;
}
```

La data di nascita è espressa nella forma `aaaammgg` per cui risulta essere ordinabile. Le prime quattro cifre rappresentano l'anno di nascita, seguono poi il mese ed il giorno di nascita.

Si noti come non vi sia conflitto fra la `classe` nome di variabile e la `classe` nome di campo. Inoltre, pur avendolo usato per brevità, si sconsiglia l'uso della costante 30 nella condizione booleana del `for`, dato che sarebbe più elegante e più corretto usare una costante mnemonica.

Il motivo di una tale scelta è già stato illustrato nella sezione 2.5 a pagina 118, per cui non si ritiene necessario ridiscutere l'argomento.

12.3 Strutture e puntatori

L'uso contemporaneo di strutture e puntatori implica una diversa notazione sintattica. In presenza di strutture puntate da puntatori, si abbandona la *dot notation* per una notazione sintattica che utilizza la freccia destra.[2]

Il seguente è un esempio di come si possano utilizzare i puntatori per puntare a delle strutture:

Figura 12.5: Esempio di uso dei puntatori nelle strutture

Si nota che se la struttura è puntata da un puntatore (`s` nell'esempio) il simbolo utilizzato per accedere il campo è la freccia `->` e non il punto.

12.3.1 Un esempio d'uso delle strutture e dei puntatori

Nella presente sezione si illustrerà un semplice esempio di uso delle strutture e dei puntatori e di come, in certi casi, problemi apparentemente complessi possano essere risolti in maniera estremamente semplice.

Esercizio - ◊◊◊ Ordinamento di strutture

Sia dato un vettore di strutture. Si vuole stampare il contenuto del vettore ordinando le strutture secondo criteri via via diversi. Si supponga, a tal fine, la porzione di codice 12.1 nella pagina precedente, e di voler ordinare dette strutture per ordine di età degli studenti.

Soluzione

La soluzione più comoda è creare un vettore di puntatori come evidenziato nel codice seguente:

Listing 12.2: Vettore di puntatori

```
Anagrafica *pa[30];
```

[2] Tale notazione, pur utilizzando una freccia, non può, per analogia, chiamarsi *arrow notation*, essendo tale dicitura già usata per una particolare notazione sintattica inventata da Donald Knuth e chiamata, appunto, *arrow notation* oppure *up-arrow notation* ed è usata per rappresentare solitamente numeri particolarmente grandi.

Abbinando ciascuna struttura ad un puntatore si potrebbero *ordinare i puntatori* anziché le strutture. Tale scelta permetterebbe di risparmiare molto tempo e un po' di memoria, non dovendo spostare intere strutture ma solamente dei semplici puntatori. Spostare la struttura, infatti, implica la copia e la memorizzazione di ciascun suo singolo campo, con notevole spreco di tempo.

Un esempio di un siffatto ordinamento è illustrato nel codice sottostante[3]

Listing 12.3: Ordinamento di puntatori

```
#include "stdio.h"

// Definizione di costanti
#define kStrLen 30      /* Massima lunghezza stringa    */
#define kSize   30      /* Massimo numero allievi       */

// Definizione della struttura Anagrafica. Detta
// struttura identifica compiutamente un allievo.
struct Anagrafica
{
   char cognome[kStrLen];
   char nome[kStrLen];
   unsigned long int dataNascita;
   char classe[kStrLen];
   unsigned char numRegistro;
};

// Funzione di immissione di numero intero. I due parametri
// della funzione rappresentano il limite inferiore e quello
// superiore (compresi) del numero da digitare.
int Immissione(int inf, int sup)
{
   int n;

   // Ciclo di immissione del numero. Se il numero digitato
   // e' fuori dai limiti, l'utente e' invitato a ridigitare
   // il numero.
   do
   {
      scanf("%d", &n);
      if(n<inf||n>sup)
         printf("Valore␣fuori␣dai␣limiti.␣Ridigitare:␣");
   }while(n<inf||n>sup);

   return n;
}



// Funzione di verifica di anno bisestile
bool LeapYear(int anno)
{
```

[3]L'ordinamento è eseguito mediante *bubble sort*. Tale algoritmo è illustrato nel capito dedicato agli ordinamenti, per cui si rimanda a tali pagine per un'analisi dettagliata e per i relativi diagrammi di flusso.

```
        if((anno%4==0&&anno%100!=0)||(anno%400==0&&anno%4000!=0))
                return true;
        else
                return false;
}



// Funzione di validazione della data
bool ValDate(int data)
{
        int anno;
        int mese;
        int giorno;

        // Calcola il giorno, mese e anno.
        giorno = data%100;
        data /= 100;
        mese = data%100;
        data /= 100;
        anno = data;

        // Valida la data
        switch(mese)
        {
                case 2: if(giorno>28) return LeapYear(anno);
                case 4:
                case 6:
                case 9:
                case 11: if(giorno>30) return false;
                default: return true;
        }
}

int Compare(char *s1, char *s2)
{
        int n=0;
        int m=0;

        while((s1[n]!='\0')&&(s2[m]!='\0'))
        {
                // Salta gli spazi e gli apostrofi nei cognomi
                if((s1[n]==' ')||(s1[n]=='\'')) n++;
                if((s2[m]==' ')||(s2[m]=='\'')) m++;

                // Confronta
                if(s1[n]>s2[m])
                        return 2;               // La stringa 2 precede la 1
                else
                {
                        if(s1[n]<s2[m])
                                return 1;       // La stringa 1 precede la 2
                        else
                        {
```

```
                                // Le due stringhe, per il momento, sono
                                // uguali: passa al prossimo carattere.
                                n++;
                                m++;
                        }
                }
        }

        // Verifica l'esito del confronto
        if((s1[n]=='\0')&&(s2[m]!='\0'))
                return 1;               // La stringa 1 precede la 2
        else
        {
                if((s1[n]!='\0')&&(s2[m]=='\0'))
                        return 2;       // La stringa 2 precede la 1
                else
                {
                        if((s1[n]=='\0')&&(s2[m]=='\0'))
                                return 0;// Le stringhe sono uguali
                }
        }
}

void Sort(Anagrafica *a[], int num)
{
        Anagrafica *t;
        int i=0;
        int r;

        while(i<num-1)
        {
                r = Compare(a[i]->cognome, a[i+1]->cognome);
                if(r==2)
                {
                        t = a[i];
                        a[i] = a[i+1];
                        a[i+1] = t;
                        i=0;
                }
                else
                        i++;
        }
}



void Upper(Anagrafica *a, int num)
{
        char *c;

        for(int j=0; j<num; j++)
        {
                // Riduci il cognome a tutte maiuscole
                c = a->cognome;
```

```
                do if(*c>='a'&&*c<='z') *c-='a'-'A'; while(*c++!='\0');

                // Riduci il nome a tutte maiuscole
                c = a->nome;
                do if(*c>='a'&&*c<='z') *c-='a'-'A'; while(*c++!='\0');
                a++;
        }
}



void Stampa(Anagrafica *a[], int num)
{
        printf("\n");
        printf("\n");
        for(int j=0; j<num; j++)
        {
                // Aggiunge il numero di registro
                a[j]->numRegistro = j+1;

                // Stampa il cognome e il nome
                printf("%d)␣%s␣%s\n", j+1, a[j]->cognome, a[j]->nome);
        }
}



int _tmain(int argc, _TCHAR* argv[])
{
   Anagrafica classe[kSize];
   Anagrafica *pa[kSize];
   int i;

   // Chiede all'utente di inserire la classe comune
   printf("Inserire␣la␣classe:␣");
   scanf("%s", classe[0].classe);
   printf("\n");

   // Chiede all'utente di inserire il numero di allievi. Si suppone
   // che la classe non possa vere meno di 3 allievi e non piu'
   // di 30.
   printf("Inserire␣il␣numero␣di␣allievi:␣");
   num = Immissione(3, 30);

   // Chiede all'utente di inserire le anagrafiche degli allievi. Non
   // viene chiesto il nome della classe perche' gia' noto.
        for(i=0; i<kSize;i++)
        {
                // Copia la classe
                //classe[i].classe = classe[0].classe;

                // Chiede all'utente l'immissione del cognome
                printf("Allievo␣N.%d\n", i+1);
                printf("Inserire␣il␣cognome␣dell'allievo:␣");
```

```
            scanf("␣%[^\n]s", classe[i].cognome);

            // Chiede all'utente di inserire il nome
            printf("Inserire␣il␣nome␣dell'allievo:␣");
            scanf("␣%[^\n]s", classe[i].nome);

            // Chiede all'utente di inserire la data di nascita
            do
            {
                    printf("Data␣di␣nascita\n");
                    printf("Anno␣(aaaa):␣");
                    classe[i].dataNascita = Immissione(1990, 2000)*100*100;
                    printf("Mese␣(mm):␣");
                    classe[i].dataNascita += Immissione(1, 12)*100;
                    printf("Giorno␣(gg):␣");
                    classe[i].dataNascita += Immissione(1, 31);
            }while(!ValDate(classe[i].dataNascita));
            printf("\n");
    }

    // Assegna ai puntatori le singole strutture
    for(i=0; i<kSize; i++)
            pa[i] = &classe[i];

    // Trasforma le stringhe in tutte maiuscole
    Upper(pa[0], kSize);

    // Ordina alfabeticamente gli allievi
    Sort(pa, kSize);

    // Aggiunge il numero di registro e
    // stampa l'elenco degli allievi
    Stampa(pa, kSize);

    return 0;
}
```

12.4 Strutture e funzioni

12.5 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 12.

Parte IV

PROGRAMMAZIONE AVANZATA

Capitolo 13

La ricorsione

In informatica viene detto *ricorsivo* un algoritmo espresso in termini di se stesso. Analogo concetto si può ritrovare in matematica come pure nelle arti figurative, come la sottostante immagine suggerisce. Se ben utilizzato la ricorsione

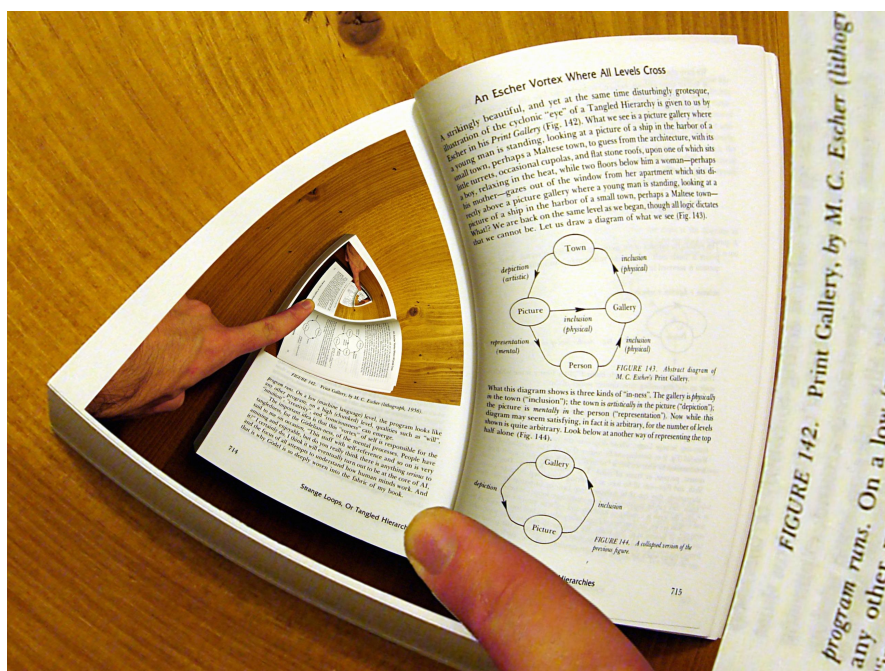



Figura 13.1: Immagine ricorsiva

è uno strumento particolarmente potente, anche se, a volte, può risultare di difficile comprensione. Diventa utile quando le azioni da eseguire su un insie-

me di dati di *input* sono ripetitive, ma mal si prestano per essere interpretate in maniera iterattiva, ossia mediante un ciclo.

Un esempio che frequentemente si utilizza per illustrare la ricorsione è quello del calcolo fattoriale. Come già visto nella sezione 4.3.2 a pagina 146, per definizione si ha

$$0! = 1 \tag{13.1}$$

dove 0! si legge *fattoriale di zero* oppure *zero fattoriale*. I fattoriali dei naturali superiori a zero assumono i seguenti valori:

$$\begin{aligned}
1! &= 1 \\
2! &= 1 \cdot 2 = 2 \\
3! &= 1 \cdot 2 \cdot 3 = 6 \\
4! &= 1 \cdot 2 \cdot 3 \cdot 4 = 24 \\
5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \\
&\dots
\end{aligned}$$

da cui si deduce che il fattoriale di 2 è uguale 2 per il fattoriale di 1; il fattoriale di 3 è uguale a 3 per il fattoriale di 2; il fattoriale di 4 è uguale a 4 per il fattoriale di 3, e così via. Quindi

$$n! = n \cdot (n-1)! \tag{13.2}$$

Le equazioni 13.1 e 13.2 conducono alla definizione ricorsiva di fattoriale per $n \in \mathbb{N}$:

$$n! = \begin{cases} 1, & \text{se } n = 0, \\ n \cdot (n-1)!, & \text{se } n \geq 1, \end{cases} \tag{13.3}$$

13.1 Un po' di teoria? No, grazie

La teoria della ricorsione è argomento matematico di estrema complessità che le presenti pagine non hanno assolutamente la pretesa di trattare (e nemmeno di introdurre). L'argomento è stato esplorato da illustri matematici (in ordine più o meno cronologico Grassmann, Dedekind, Hilbert, Skolem, Gödel, Peano, Kleene ed altri, infine coronato dalla monumentale opera del nostro Piergiorgio Odifreddi nel *Classical Recursion Theory* Vol. I e II), per cui si rimanda umilmente a loro per la trattazione dell'argomento, ricordando all'ignaro lettore che non è fra le più banali.

Una trattazione molto più semplice, anche se meno rigorosa, è quella proposta da Niklaus Wirth, introducendo un punto di vista più informatico.

In tale ambito la funzione ricorsiva F è rappresentata da un insieme $\mathcal{P}$ di istruzioni, costituite a loro volta da S_i istruzioni (non contenenti F) e da F, per cui essa è definita come

$$F \equiv \mathcal{P}[S_i, F] \tag{13.4}$$

Una siffatta funzione presenta, però, un problema: i calcoli che essa esegue potrebbero non terminare mai. Diventa quindi importante introdurre il concetto di *terminazione* della funzione, ovvero determinare per quali condizioni la funzione termina l'esecuzione delle computazioni.

L'espressione 13.4 va quindi arricchita in modo da evidenziare la *condizione di terminazione* della computazione, come in 13.5 oppure in 13.6

$$F \equiv \mathcal{P}[\textbf{se } B \textbf{ allora } S_i \textbf{ altrimenti } F] \tag{13.5}$$

$$F \equiv \mathcal{P}[S_i, \textbf{se } B \textbf{ allora } F] \tag{13.6}$$

dove B rappresenta proprio la condizione di terminazione della ricorsione o *caso base*, mentre F, richiamato all'interno della funzione, è detto *caso induttivo*. Le due espressioni non sono equivalenti, ma insieme coprono i diversi contesti operativi che si possono incontrare come pure le diverse soluzioni adottate. Naturalmente le due condizioni di terminazione B non è detto che siano uguali.

Le 13.5 e 13.6 rappresentano due diverse definizioni simboliche di *ricorsione diretta*. Essa si ottiene quando la funzione F è direttamente richiamata all'interno della stessa funzione F. Esiste anche la possibilità che la funzione F richiami la funzione Q che, a sua volta, richiami la funzione F. In questo caso si parla di *ricorsione indiretta*.

Le due espressioni 13.5 e 13.6 possono essere ulteriormente sviluppate e rappresentare ancora meglio la realtà computazionale. Ciò nasce dal fatto che l'argomento della funzione è un intero, per cui la condizione di terminazione è legata proprio all'intero passato in argomento, come pure i casi induttivi che seguono, per cui le espressioni che evidenziano tali legami potrebbero essere le seguenti:

$$F(n) \equiv \mathcal{P}[\textbf{se } n < n_0 \textbf{ allora } S_i \textbf{ altrimenti } F(n-1)] \tag{13.7}$$

$$F(n) \equiv \mathcal{P}[S_i, \textbf{se } n > n_0 \textbf{ allora } F(n-1)] \tag{13.8}$$

Le due espressioni 13.7 e 13.8 ricoprono un ruolo fondamentale nel guidare lo studente verso una corretta impostazione di funzioni ricorsive. Anche esse non sono però esaustive e non coprono la totalità dei problemi ricorsivi che si possono presentare. Infatti nelle due espressioni 13.7 e 13.8 il valore assunto da $F(n)$ dipende solamente dalla legge di successione $\mathcal{P}$ che lega il valore di un elemento a quello successivo e dal valore di $F(n-1)$.

In realtà si possono immaginare funzioni ricorsive che elaborano il valore di $F(n)$ partendo dai precedenti k elementi, con $k \in \mathbb{N}|k > 1$. Un esempio classico per $k = 2$ è dato dalla successione di Fibonacci, già incontrata nei capitoli precedenti e valida per $n \in \mathbb{N}$:

$$f(n) = \begin{cases} 0, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ f(n-1) + f(n-2), & \text{se } n \geq 2. \end{cases} \tag{13.9}$$

In tal caso, le 13.7 e 13.8 potrebbero essere modificate in:

$$F(n) \equiv \mathcal{P}[\textbf{se } n < n_0 \textbf{ allora } S_i \textbf{ altrimenti } F(m_1, m_2..)] \tag{13.10}$$

$$F(n) \equiv \mathcal{P}[S_i, \textbf{se } n > n_0 \textbf{ allora } F(m_1, m_2..)] \tag{13.11}$$

con $m_1, m_2.. < n$.

Alla luce di quanto fin qui detto è possibile illustrare due esempi pratici con $k = 1$ e $k = 2$ e correlare le espressioni elaborate con detti esempi. Il primo dei due esempi è la funzione `Fat`, che calcola ricorsivamente il fattoriale di un numero intero dato in argomento. In questo caso il successore è calcolabile utilizzando una sola chiamata a funzione, per cui $k = 1$. La funzione è simile alla seguente:

```
int Fat(int n)
{
  if(n==0)
    return 1;
  else
    return n*Fat(n-1);
}
```


Figura 13.2: Esempio di funzione ricorsiva a una chiamata

L'insieme $\mathcal{P}$ di istruzioni è dato dall'intero corpo della funzione; la condizione di terminazione è il test se n vale zero; l'istruzione eseguita se la condizione di terminazione è verificata è il ritorno del valore noto della successione, ossia del valore del primo elemento; la ricorsione vera e propria è data dal richiamo della funzione `Fat` all'interno della `Fat` stessa, più precisamente è data dall corpo dell'`else`. La struttura della funzione è del tutto simile alla 13.7.

Il secondo esempio è la funzione `Fibo`, che calcola ricorsivamente il valore dell'ennesimo elemento della successione di Fibonacci. In questo caso il successore è calcolabile utilizzando due chiamate a funzione, per cui $k = 2$. La funzione è simile alla seguente:

```
int Fibo(int n)
{
  if(n==0||n==1)
    return n;
  else
    return Fibo(n-1)+Fibo(n-2);
}
```


Figura 13.3: Esempio di funzione ricorsiva a due chiamate

Ora si possono fornire allo studente due semplici regole pratiche per scrivere una funzione ricorsiva:

1. la metodo ricorsivo deve sempre ritornare direttamente un valore che la funzione assume **in almeno un caso particolare**. Si tratta del gruppo di istruzioni indicate col simbolo S_i nelle funzioni `Fat` e `Fibo`. Si noti che i casi particolari possono essere più d'uno. Nel caso della funzione `Fibo`, ad esempio, i casi particolari (o meglio i *casi base*) sono due;
2. il caso induttivo va richiamato **dopo aver semplificato il problema**. Nel presente caso "semplificare" significa avvicinarsi ad uno dei casi base. Se, ad esempio, il caso base prevede la condizione `n==0` e la funzione ha per parametro `n` con `n>0`, allora il caso induttivo va richiamato **diminuendo** n, in modo da avvicinarsi al caso base. Se il problema non viene semplificato il caso induttivo viene richiamato all'infinito creando, prima o poi, un errore di *stack overflow*.

13.1.1 Quando e perché (non) usare la ricorsione

Il "non" tra parentesi vuol significare che si parlerà di quando usare e quando non usare la ricorsione. Dalle precedenti pagine si apprende che la ricorsione presenta più di un problema. Perché usarla? E quando? E, viceversa, quando non usarla?

Naturalmente l'ultima parola spetta al programmatore, che però deve essere messo nelle condizioni di decidere a ragion veduta. In termini generali, se un problema dato si può agevolmente risolvere mediante un metodo iterattivo è bene non ricorrere alla ricorsione. I motivi che spingono a decidere per il metodo iterattivo sono essenzialmente legati alla velocità di esecuzione dell'algoritmo: il metodo ricorsivo è solitamente molto più lento.

Un secondo difetto della ricorsione è dato dall'uso ingordo di memoria *stack*.[1] Ogni chiamata induttiva significa nuova allocazione di memoria nello *stack*. Ciò è ovvio, trattandosi di una chiamata a funzione: si deve memorizzare nello *stack* l'indirizzo di ritorno della funzione, i suoi parametri e le sue variabili locali. Se la funzione viene richiamata 100 volte, 100 volte si deve allocare memoria per gli oggetti predetti.

Ma allora quando conviene utilizzare il metodo ricorsivo? Essenzialmente quando il problema è già definito in maniera ricorsiva e risulta difficile ridefinirlo in maniera non ricorsiva oppure quando la soluzione iterattiva di un problema si presenta essere sostanzialmente difficile e il problema, pur non essendo esposto in maniera ricorsiva, *appare* essere un problema ricorsivo.

13.1.2 Il tempo di esecuzione

Prima si utilizzare una funzione ricorsiva è bene valutare seriamente la possibilità di risolvere il problema in maniera iterattiva. Se ciò non è possibile o risulta difficile si attuerà il metodo ricorsivo, tenendo bene a mente, però, che ciò comporta uno spreco di tempo e di memoria. Nella presente sezione si vuole comparare una soluzione iterattiva con quella ricorsiva e valutare quantitativamente e qualitativamente la differenza fra le due soluzioni. La comparazione verrà effettuata sia sul fronte del tempo di esecuzione che della memoria allocata.

Si valuti, ad esempio, il seguente codice iterattivo:

[1] Si è già accennato all'argomento. Esso verrà trattato esaustivamente in altro capitolo.

Listing 13.1: Fibonacci iterattivo

```
int FiboIter(int n)
{
   int i;
   int f0, f1, f2;

   if((n==0)||(n==1))
      return n;
   else
      for(i=1, f0=0, f1=1; i<=n; i++)
      {
         f2 = f1+f0;
         f0=f1;
         f1=f2;
      }
      return f2;
}
```

La memoria occupata dalla suddetta chiamata a funzione è data da:

- 4 byte relativi al valore di ritorno;
- 4 byte relativi all'argomento;
- 4x4=16 byte relativi alle variabili locali;
- 4 byte di indirizzo di ritorno.

In totale vengono allocati 28 byte di memoria *indipendentemente* dal valore di n.

Oltre all'occupazione di memoria si deve valutare anche il tempo impiegato per eseguire le singole istruzioni. Siccome non è dato sapere a quanto ammontano quantitativamente i singoli tempi di esecuzione, si quantifica il totale nel seguente modo:

- la chiamata a funzione, compresa l'allocazione di memoria, occupa il tempo t_1
- l'inizializzazione delle variabili `i`, `f0` e `f1` occupano il tempo t_2;
- l'esecuzione della prima istruzione del corpo del ciclo occupa il tempo $n \cdot t_3$;
- l'esecuzione della seconda più la terza istruzione del corpo del ciclo occupa il tempo $n \cdot t_4$;
- l'esecuzione della condizione di terminazione del ciclo occupa il tempo $n \cdot t_5$;
- l'esecuzione del caso base occupa il tempo t_6;
- il `return` occupa il tempo t_7. Si noti a tal proposito che non ha senso calcolare il tempo occorrente per eseguire il `return` due volte (la `return n` e la `return f2`), dato che se viene eseguita una delle due, l'altra non può più essere eseguita.

In totale si ha

$$t_{TOT} \simeq t_1 + t_2 + t_6 + t_7 + n \cdot (t_3 + t_4 + t_5) \tag{13.12}$$

Si tenga presente che dei 7 tempi citati, il primo, ossia il tempo t_1 è quello predominante, dato che deve comprendere la ricerca di adeguato spazio libero in memoria e l'allocazione di detta memoria.

La funzione ricorsiva assume la seguente forma:

Listing 13.2: Fibonacci ricorsivo

```
int FiboRico(int n)
{
   if((n==0)||(n==1))
      return n;
   else
      return FiboRico(n-1)+FiboRico(n-2);
}
```

Apparentemente l'occupazione di memoria sembra minore, dato che mancano le variabili locali. Però la funzione alloca $28 - 16 = 12$ byte *ad ogni chiamata di funzione*. Per $n = 20$ la funzione viene richiamata ben 13529 volte, per un totale di 162348 byte, contro i 28 della funzione iterattiva. La differenza è spaventosa e aumenta rapidamente all'aumentare di n.

Ragionamento analogo si può fare per il calcolo del tempo. Supponendo simili i tempi di calcolo dei valori $n - 1$ e $n - 2$ al tempo t_3 e siccome usando il metodo ricorsivo la funzione, per $n = 20$ viene richiamata 13529 volte, si possono quantificare i tempi nel seguente modo:

$$t_{TOT} \simeq 13529 \cdot (t_1 + t_6 + t_7 + t_3) \tag{13.13}$$

Anche in questo caso la differenza è assolutamente rilevante e lo diventa in maniera sempre più evidente man mano che il valore di n aumenta. Resta da illustrare come si sia giunti al numero 13529. Un grafico può aiutare a capire come si sviluppano le chiamate alla funzione.

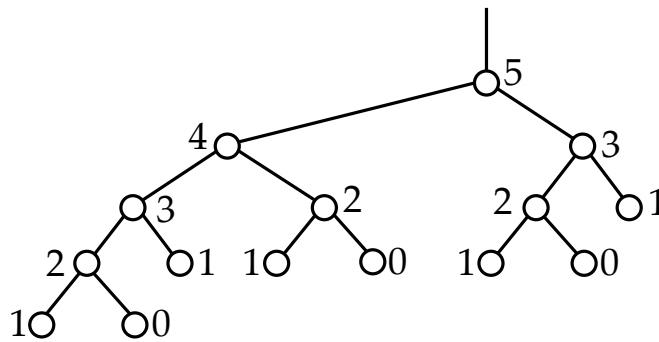



Figura 13.4: Albero delle chiamate di Fibonacci per $n = 5$

Se si richiama Fibonacci ricorsivamente con $n = 5$, si richiama la stessa funzione con $n = 4$ e $n = 3$. Percorrendo due rami così generati si nota che richiamando, da un lato, Fibonacci con $n = 4$, si richiama la funzione altre due volte con $n = 3$ e $n = 2$. Dall'altro lato si richiamerà Fibonacci con $n = 2$ e $n = 1$ e così via finché tutti i rami dell'albero terminano con $n = 1$ oppure con $n = 0$, ovvero con le due condizioni di terminazione. Il suddetto grafico conta, in totale 15 nodi.

In maniera del tutto analoga si possono tracciare i grafici per $n = 0, 1, 2, 3...$ ecc. Naturalmente è impensabile tracciare il grafico per $n = 20$, dato l'altissimo

numero di nodi. Si tratta, quindi, di rilevare la legge che lega il valore di un nodo ai due precedenti.

Disegnando diversi grafici si ottengono i seguenti numeri:

n	Chiamate
0	1
1	1
2	3
3	5
4	9
5	15
6	25

Tabella 13.1: Relazione fra n ed il numero di chiamate a funzione

Si nota piuttosto facilmente che per $n \geq 2$ il valore delle chiamate è dato dalla somma del numero delle due chiamate precedenti più 1. Infatti, anche la relazione che lega il numero delle chiamate a funzione p ad n può essere formulata in modo ricorsivo:

$$p(n) = \begin{cases} 1, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ p(n-1) + p(n-2) + 1, & \text{se } n \geq 2. \end{cases} \tag{13.14}$$

La successione che si ottiene per i primi 20 elementi è quindi la seguente:

1 1 3 5 9 15 25 41 67 109 177 287 465 753 1219 1973 3193 5167 8361 13529

Si noti che la successione di Fibonacci cresce in maniera meno rapida:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

I numeri fin qui evidenziati dovrebbero quindi consigliare prudenza nell'uso della ricorsione. In taluni casi, comunque, la ricorsione si rivela uno strumento assolutamente potente ed utilissimo, come si vedrà nei prossimi esercizi. Prima, però, si proporranno alcuni esercizi molto facili per "allenare" lo studente all'uso di tale strumento. Si tratta di esercizi che si risolvono in maniera iterattiva in modo molto semplice, ma di cui si consiglia, per i motivi predetti, lo svolgimento ricorsivo.

13.1.3 Ma perché funziona?

Frequentemente lo studente non si capacita del perché la ricorsione funzioni. In tal caso lo studente non ha capito *come* funziona. Cercando di interpretare un comune dubbio, si dedica una apposita sezione all'argomento. Si cercherà di illustrare il meccanismo di elaborazione della funzione ricorsiva utilizzando un esempio ormai trito, ossia il calcolo del fattoriale. Il motivo è sempre lo stesso: non sommare dubbio a dubbio. Per comodità espositiva si ripropone il codice di detta funzione:

Listing 13.3: Funzione di calcolo fattoriale

```
int Fat(int n)
{
    if(n==0)
        return 1;
    else
        return n*Fat(n-1);
}
```

Si supponga la condizione iniziale $n = 3$ (vedi punto 1 di fig. 13.5). Cosa succede quando verrà eseguita la funzione `Fat` per la prima volta con il parametro posto al valore 3? Verrà testata inizialmente la condizione di terminazione con $n = 3$ e risulterà falsa. Verrà quindi eseguito il corpo dell'`else` dove si moltiplicherà 3 per il valore ritornato dalla funzione `Fat` con argomento posto a 2. Ciò significa che si richiamrà per la seconda volta la funzione `Fat`, questa volta con argomento posto a 2 (vedi punto 2 di fig. 13.5). Siccome la funzione richiamata per la seconda volta non è ancora terminata, la predetta moltiplicazione rimarrà sospesa in attesa del valore ritornato.

Dal punto di vista grafico la situazione è la seguente:

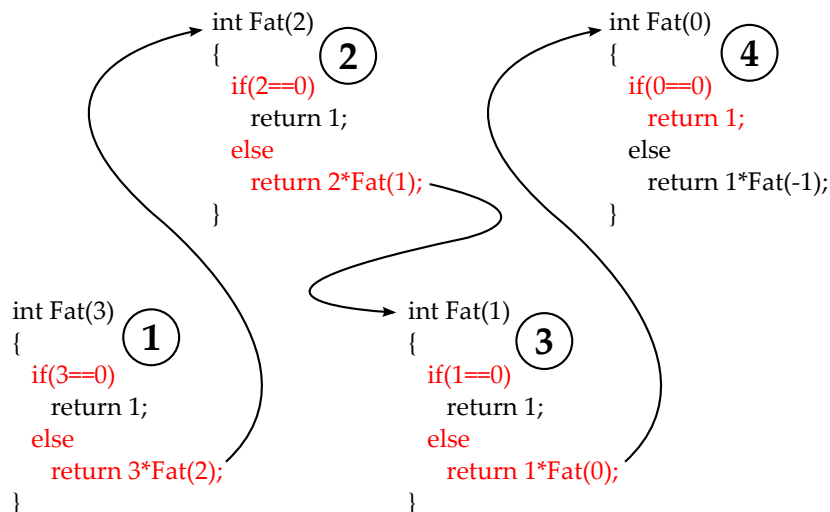



Figura 13.5: Calcolo effettivo del fattoriale

Entrati per la seconda volta nella funzione `Fat` verrà testata nuovamente la condizione di terminazione, e siccome risulterà falsa verrà eseguito il corpo dell'`else` anche nella seconda funzione. Si noti a tal proposito che il primo richiamo di `Fat` ed il secondo richiamo **utilizzano variabili differenti e totalmente indipendenti**. Anche nel caso 2 la moltiplicazione verrà sospesa, perché la funzione `Fat` con argomento 1 non è stata ancora eseguita.

Si entrerà quindi per la terza volta nella funzione `Fat`, stavolta con argomento 1 (vedi punto 3 di fig. 13.5). Anche in questo caso, però, si ripresenterà lo stesso scenario: la condizione di terminazione verrà considerata falsa e la moltiplicazione del corpo dell'`else` resterà sospesa. Contestualmente verrà richiamata la funzione `Fat` con argomento 0 (vedi punto 4 di fig. 13.5).

Richiamando la funzione `Fat` con argomento 0, finalmente la condizione di terminazione risulterà vera e verrà eseguito il corpo dell'`if`. Verrà cioè eseguito il caso base ritornando un valore effettivo, ossia 1. Tale valore permetterà il completamento dell'ultima moltiplicazione lasciata in sospeso (`1*Fat(0)` alla fine del punto 3 di fig. 13.5), per cui verrà ritornato il valore 1.

In tal modo potrà essere completata anche la seconda moltiplicazione lasciata in sospeso, posta alla fine del punto 2 di fig. 13.5 ($2 \cdot 1 = 2$). La funzione potrà ritornare il valore 2 e permettere l'esecuzione della prima moltiplicazione lasciata in sospeso, ossia $3 \cdot 2 = 6$. Calcolato detto valore, la funzione potrà restituire il risultato definitivo del calcolo.

13.2 La ricorsione attraverso gli esercizi

Si è fin qui cercato di far convivere nella giusta misura teoria e pratica, in modo da non svilire eccessivamente il costrutto teorico della ricorsione (anche tenendo conto delle basi matematiche dello studente di scuola superiore), ma dando il giusto peso anche agli esempi pratici.

Nella presente sezione verranno presentati alcuni esempi svolti di problemi legati al metodo ricorsivo. In taluni casi la soluzione ricorsiva apparirà piuttosto forzata, essendo la soluzione iterattiva evidente e fondamentalmente più semplice. Si è deciso in tal senso sempre per lo stesso motivo: proporre allo studente un problema alla volta. Si ritiene quindi didatticamente più appropriato permettere allo studente di concentrarsi sul metodo piuttosto che sulla difficoltà dell'esercizio. Per tal motivo i primi esercizi proposti saranno molto familiari allo studente ed obiettivamente piuttosto semplici.

Gli ultimi esercizi saranno più genuinamente ricorsivi e *dovranno* essere trattati come tali. Si spera che nel frattempo lo studente abbia acquisito sufficiente confidenza con lo strumento della ricorsione da potersi concentrare su problemi più difficili. Come al solito, si invita lo studente a tentare di svolgere da solo l'esercizio proposto e di leggere e studiare la soluzione solo in un secondo momento.

13.2.1 Sommatoria ricorsiva

Il presente esercizio serve solo a "scaldare i muscoli" e permettere allo studente di iniziare a prendere confidenza con il metodo ricorsivo.

Esercizio - ◊◊◊ Sommatoria ricorsiva

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dato un intero non negativo n posto a parametro, ritorni la sommatoria da 0 a n di k, ossia

$$\sum_{k=0}^{n} k \tag{13.15}$$

Soluzione

La difficoltà dell'esercizio consiste nel riformulare il problema in maniera ricorsiva, per cui non si chiede la stesura di un diagramma di flusso, che sarebbe di scarso aiuto, anche semplicemente in termini documentativi.

L'attenzione deve quindi essere posta sul caso base e sul caso induttivo. Riuscire a definire con chiarezza la condizione di terminazione e definire il caso induttivo in modo tale da avvicinarsi via via al caso base, significa ridefinire il problema in termini ricorsivi.

Il caso base è chiaramente identificato dalla condizione $n = 0$: se il parametro della funzione assume il valore 0, è immediatamente possibile stabilire il valore della sommatoria, che vale ovviamente 0.

Il caso induttivo non è molto più complicato: esso va formulato per $n \geq 1$ in modo tale da richiamare la stessa funzione con il parametro posto ad un valore tale da avvicinarsi al caso base. Nel presente esempio, trattandosi di una sommatoria di numeri naturali successivi, sembra abbastanza evidente che il parametro debba essere decrementato di 1.

Potremmo quindi enunciare ricorsivamente il problema nel seguente modo:

$$s(n) = \begin{cases} 0, & \text{se } n = 0, \\ n + s(n-1), & \text{se } n \geq 1. \end{cases} \tag{13.16}$$

Una volta definito ricorsivamente il problema si può passare alla scrittura del codice, che deve rispecchiare fedelmente la definizione ricorsiva. La necessità di ridefinire ricorsivamente il problema sta proprio qui: una volta ridefinito in termini ricorsi il problema dato ci si deve attenere strettamente alla definizione per scrivere il codice.

Listing 13.4: Sommatoria ricorsiva

```
int Sum(int n)
{
   if(n==0)      //Condizione di terminazione
      return 0; //Assegnazione caso base
   else
      return n+Sum(n-1); //Caso induttivo
}
```

La sottostante funzione di test prevede l'uso della funzione `DefIntNum`. Nei prossimi esercizi la si darà per scontata.

Listing 13.5: Test sommatoria

```
int DefIntNum(int liminf, int limsup)
{
   int m;

   //Ciclo di inserimento del numero richiesto.
   //Si tratta di un ciclo do..while.
   do
   {
      //Frase di cortesia. Viene scritta su due righe per
      //motivi grafici imputabili alla presente pagina.
      printf("Digitare␣un␣numero␣compreso␣fra␣%d", liminf);
      printf("␣e␣%d:␣", limsup);
      scanf("%d", &m);
   }while(m<liminf||m>limsup);
   return m;    //Ritorna il numero
}
```

```
int main(void)
{
   const int kLim = 100; //Si pone il limite superiore a 100
   int s;                //Sommatoria

   s = DefIntNum(0, kLim);
   printf("La␣sommatoria␣da␣0␣a␣%d␣vale␣%d", kLim, Sum(s));
   return 0;
}
```

13.2.2 Potenza ricorsiva

Anche il presente esercizio è puramente introduttivo. Sarebbe più saggio scrivere il codice in forma iterattiva anziché ricorsiva. Lo studente ricordi, però, la funzione attribuita ai primi esercizi.

Esercizio - ◊◊◊ Potenza ricorsiva

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dati due interi $b, e \in \mathbb{N} | b > 0, e \geq 0$ posti a parametro, ritorni b^e.

Soluzione

Dobbiamo sempre riformulare il problema in termini ricorsivi. Dobbiamo cioè definire con chiarezza il caso base e d il caso induttivo. Quest'ultimo deve essere definito in modo tale da avvicinare via via il valore dell'argomento (o degli argomenti) alla condizione di terminazione.

Nel presente caso gli argomenti della funzione sono due: una base b ed un esponente e. La seconda delle due "regole" che si dovrebbero seguire per definire efficientemente un algoritmo ricorsivo consiglia di "avvicinare" l'argomento (gli argomenti) del caso induttivo alla condizione di terminazione. La domanda che ci si pone è: quale dei due argomenti va "avvicinato"? Entrambi?

Per rispondere è necessario riflettere sul concetto di elevazione a potenza. Eseguire il calcolo b^e significa moltiplicare l'elemento neutro della moltiplicazione (ossia 1) per b tante volte quanto vale e. Se $e = 0$ si moltiplica l'unità per b zero volte, per cui il risultato è 1. Se $e = 1$ si moltiplica l'unità per b una volta, per cui il risultato è b. Se $e = 2$ si moltiplica l'unità per b due volte, per cui il risultato è $b \cdot b$ e così via.

Sembrerebbe, quindi, che il caso base e il caso induttivo siano legati più ad e che a b. In particolare, ipotizzando il caso base $e = 0$ siamo assolutamente in grado di fornire un risultato immediato, cioè 1 e ciò indipendentemente da b.

Una volta formulato il caso base, si deve formulare il caso induttivo. Per $e > 0$ l'elevamento a potenza può essere definito come il prodotto $b \cdot b^{e-1}$. Si potrebbe quindi riformulare ricorsivamente l'elevamento a potenza nel seguente modo:

$$f(b,e) = \begin{cases} 1, & \text{se } e = 0, \\ b \cdot f(b, e-1), & \text{se } e \geq 1. \end{cases} \tag{13.17}$$

oppure, se si preferisce, in maniera ancora più esplicita nel seguente modo:

$$b^e = \begin{cases} 1, & \text{se } e = 0, \\ b \cdot b^{e-1}, & \text{se } e \geq 1. \end{cases} \tag{13.18}$$

In entrambe le espressioni il caso base ed il caso induttivo sono chiaramente espressi, per cui si può tentare di scrivere il codice della funzione ricorsiva attenendosi rigorosamente alla 13.17 o alla 13.18:

Listing 13.6: Elevamento a potenza ricorsivo

```
int Pot(int b, int e)
{
   if(e==0)      //Condizione di terminazione
      return 1; //Assegnazione caso base
   else
      return b*Pot(e-1); //Caso induttivo
}
```

Il programma di test è assolutamente simile a quello proposto nella sezione precedente (la funzione `DefIntNum` non è esplicitata):

Listing 13.7: Test elevamento a potenza

```
int main(void)
{
   const int kLim = 9;  //Si pone il limite superiore a 9
   int b, e;            //Base ed esponente

   b = DefIntNum(1, kLim);
   e = DefIntNum(0, kLim);
   printf("%d␣elevato␣alla␣%d␣e'␣uguale␣a␣%d", b, e, Pot(b,e));
   return 0;
}
```

13.2.3 MCD ricorsivo

Il presente esercizio richiede già un maggior sforzo da parte dello studente. Sia il caso base che quello induttivo non sono totalmente intuitivi. Lo si studi quindi attentamente.

Esercizio - ◊◊♦ MCD ricorsivo

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dati due interi $m, n \in \mathbb{N} | m > n > 0$ posti a parametro, ritorni il MCD dei due interi mediante l'algoritmo di Euclide.

Soluzione

Per coloro i quali non si dovessero ricordare l'algoritmo di Euclide per il calcolo del MCD fra due interi $m, n \in \mathbb{N} | m > n > 0$, lo si espone brevemente:

1. si calcola il resto r della divisione intera m/n;
2. se $r = 0$ l'algoritmo termina e il risultao è dato da n;
3. altrimenti si pone nell'ordine $m \leftarrow n$, $n \leftarrow r$ e si torna al punto 1.

Noto l'algoritmo, esso va ridefinito in maniera ricorsiva secondo le regole utilizzate precedentemente. Il caso base deve poter restituire un valore immediatamente calcolabile, mentre il caso induttivo deve richiamare lo stesso algoritmo in maniera "semplificata". A ben guardare, però, l'algoritmo originale indicato a pag. 339 è praticamente già esposto in forma ricorsiva. E' sufficiente rielaborarlo leggermente nella seguente forma:

1. se m è divisibile per n il MCD è n;
2. altrimenti calcola il MCD fra n ed il resto della divisione intera m/n.

Il punto esprime chiaramente il caso base, dato che al verificarsi della condizione di terminazione è possibile determinare un risultato in forma analitica senza bisogno di ulteriori calcoli iterattivi o ricorsioni. Il secondo punto esprime invece il caso induttivo, dato che viene richiamata la stessa funzione con un argomento "semplificato" ovvero tale da avvicinarsi alla condizione di terminazione. E' quindi possibile redarre l'algoritmo ricorsivo in forma simbolica:

$$f(m,n) = \begin{cases} n, & \text{se } m\%n = 0, \\ f(n, m\%n), & \text{se } m\%n > 0. \end{cases} \tag{13.19}$$

dove con il simbolo "%" si intende *modulo* oppure *resto della divisione intera*. La 13.19 può essere ulteriormente sviluppata e la divisione può diventare una sottrazione ripetuta, come nella sottostante definizione ricorsiva del MCD:

$$f(m,n) = \begin{cases} n, & \text{se } m = n, \\ f(m-n, n), & \text{se } m > n, \\ f(m, n-m), & \text{se } n > m. \end{cases} \tag{13.20}$$

Concettualmente le due definizioni sono identiche, solo che la prima opera delle divisioni e la seconda delle sottrazioni ripetute. Delle due la prima è semplicemente più attinente a come l'algoritmo di Euclide è esposto. Si potrebbe obiettare che la seconda definizione non necessita del vincolo $m > n > 0$, dato che si potrebbe benissimo avere $n > m > 0$. In realtà ciò vale anche per la prima definizione. Si supponga, ad esempio $m = 8$ e $n = 12$. Eseguendo la divisione intera $8/12$ si ottiene 0 come quoziente e 8 come resto, per cui nel caso induttivo seguente si ha $m \leftarrow n$ e $n \leftarrow r$ ottenendo la situazione iniziale $m = 12$ e $n = 8$.

Tornando alla definizione ricorsiva del problema, è possibile scrivere la funzione attenendosi strettamente alla definizione elaborata, ad esempio la 13.19:

Listing 13.8: MCD ricorsivo

```
int Mcd(int m, int n)
{
   if(m%n==0)   //Condizione di terminazione
      return n; //Assegnazione caso base
   else
      return Mcd(n, m%n); //Caso induttivo
}
```

Una funzione assolutamente equivalente, basata sulle sottrazioni ripetute anziché sulle divisioni è la seguente:

Listing 13.9: MCD ricorsivo con sottrazioni ripetute

```
int MCD(int m, int n)
{
   if(m==n)      //Condizione di terminazione
      return n; //Assegnazione caso base
   if(m>n)
      return Mcd(m-n, n); //Caso induttivo 1
   if(n>m)
      return Mcd(m, n-m); //Caso induttivo 2
}
```

Il programma di test è sempre simile a quelli proposti nelle sezioni precedenti. L'unica differenza è data dal fatto che nel codice 13.10 a pagina 341 sono testati entrambi i metodi, evidenziati con l'indicazione "Metodo 1" (metodo più attinente al testo dell'algoritmo, ossia mediante divisioni) e "Metodo 2" (effettuato mediante sottrazioni ripetute). Naturalmente i due risultati sono identici.

Listing 13.10: Test elevamento a potenza

```
int main(void)
{
   const int kLim = 100;
   int m,n;

   //Chiede all'utente i due numeri di cui calcolare
   // il MCD. Si noti che il secondo e' necessariamente
   //maggiore del primo. Volendo, lo si puo' far notare
   //mediante una frase di cortesia prima della seconda
   //immissione.
   n = DefIntNum(1, kLim);
   m = DefIntNum(n, kLim);

   //Stampa entrambi i risultati, con i due metodi.Il risultato
   //e' lo stesso.
   printf("Metodo␣1:␣il␣MCD␣fra␣%d␣e␣%d␣e'␣%d\n",m,n,Mcd(m,n));
   printf("Metodo␣2:␣il␣MCD␣fra␣%d␣e␣%d␣e'␣%d\n",m,n,MCD(m,n));
   return 0;
}
```

13.2.4 Serie ricorsiva

Ormai lo studente dovrebbe aver maturato una confidenza minima tale da poter affrontare il presente esercizio con una certa tranquillità. Se dovessero ancora permanere serie difficoltà nell'individuare il caso base ed il caso induttivo, potrebbe essere necessario rivedere con maggior calma la teoria presentata.

Esercizio - ◊◊◊ Serie ricorsiva

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, data la sottostante, già nota, serie, ne calcoli il risultato mediante $n+1$ termini.

$$S(n) = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + ... + \frac{1}{2^n} \tag{13.21}$$

Soluzione

Come già visto nella sezione 11.1.4 conviene riscrivere il primo termine con la notazione $1/2^0$, in modo da porre in evidenza che tutti i termini hanno una potenza di due a denominatore. La serie 13.21 diventa quindi:

$$s(n) = \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + ... + \frac{1}{2^n} \tag{13.22}$$

Si tratta ora di definire il caso base e il caso induttivo della serie proposta. Il caso base è sicuramente dato dalla seguente espressione:

$$s(0) = 1 \tag{13.23}$$

Il caso induttivo si può esprimere mediante la seguente espressione:

$$s(n) = \frac{1}{2^n} + s(n-1) \tag{13.24}$$

Quindi la serie può essere ridefinita ricorsivamente come:

$$s(n) = \begin{cases} 1, & \text{se } n = 0, \\ \frac{1}{2^n} + s(n-1), & \text{se } n \geq 1. \end{cases} \tag{13.25}$$

E' ora possibile definire il codice, semplicemente "traducendo" la nuova definizione della serie in linguaggio C:

Listing 13.11: Serie ricorsiva

```
double Serie(int n)
{
   if(n==0)       //Condizione di terminazione
      return 1.0; //Assegnazione caso base
   else
      return 1.0/Pot(2,n)+Serie(n-1); //Caso induttivo
}
```

dove la funzione `Pot` è la funzione ricorsiva atta al calcolo dell'elevamento a potenza vista nella sezione 13.2.2. La funzione di test potrebbe avere la seguente forma:

Listing 13.12: Test serie di potenze

```
int main(void)
{
   const int kLim = 20; //Limite numero di termini
   double s;            //Risultato della serie
   int n;               //Numero di termini

   //Chiede all'utente il numero di termini + 1.
   n = DefIntNum(0, kLim);

   //Calcola la serie. La funzione utilizza la Pot per
   //l'elevamento a potenza.
   printf("La␣serie␣1+1/2+1/4+1/8␣...␣con␣%d␣termini␣", n+1);
   printf("produce␣il␣seguente␣risultato:␣%f", Serie(n));
   return 0;
}
```

13.2.5 Congettura di Collatz ricorsiva

Anche la congettura di Collatz è familiare allo studente ed è in qualche modo anche già definita in maniera ricorsiva, per cui non si dovrebbero incontrare eccessive difficoltà.

Esercizio - ◊◊♦ Congettura di Collatz ricorsiva

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dato il sottostante algoritmo stampi il corrispondente valore di n ad ogni passaggio di avvicinamento al caso base:

1. sia dato un numero intero positivo n;
2. se n è dispari, si assegni ad n il risultato del calcolo $3n + 1$;
3. se n è pari si assegni ad n il risultato del calcolo $n/2$;
4. se $n = 1$ l'algoritmo termina, altrimenti si torna al punto 2.

Soluzione

La condizione di terminazione è piuttosto evidente ed è data dall'espressione $n = 1$. Anche il caso induttivo è piuttosto evidente anche se non è per nulla chiaro come si possa intendere la "semplificazione". Infatti quest'ultima è intimamente legata alla congettura: se la congettura dovesse essere errata, il caso induttivo così come è espresso nell'algoritmo sarebbe errato e la funzione sottostante non avrebbe alcun senso.

Fatte le debite premesse, siamo comunque in grado di riformulare ricorsivamente la congettura:

$$f(n) = \begin{cases} \textit{void}, & \text{se } n = 1, \\ f(n/2), & \text{se } n\%2 = 0; \\ f(3n+1), & \text{se } n\%2 = 1. \end{cases} \tag{13.26}$$

dove, come al solito, il simbolo "%" indica l'operatore di modulo e col termine *void* si intende il termine dell'algoritmo e il valore di ritorno della funzione (che è, appunto, assente).

La funzione ricorsiva può essere implementata nel seguente modo:

Listing 13.13: Congettura di Collatz ricorsiva

```
void Collatz(int n)
{
   printf("%d\n", n);     //Stampa l'argomento
   if(n==1)               //Condizione di terminazione
      return;             //Fine dell'algoritmo
   else
   {
      if(n%2==0)          //Definisce il caso induttivo
         Collatz(n/2);    //Caso induttivo 1
      else
         Collatz(3*n+1); //Caso induttivo 2
   }
}
```

Ad un'analisi meno superficiale sembra effettivamente di poter dire che il primo dei due casi induttivi sia un'evidente semplificazione dell'argomento, che avvicina, anche piuttosto rapidamente, alla condizione di terminazione. Il secondo caso induttivo allontana, però, sensibilmente dalla condizione di terminazione, per cui si può solamente sperare che la congettura sia vera.

E' possibile ora scrivere la funzione di test:

Listing 13.14: Test congettura di Collatz

```
int main(void)
{
   const int kLim = 1000; //Limite numero di partenza
   int n;                 //Numero di partenza

   //Chiede all'utente il numero di partenza.
   n = DefIntNum(1, kLim);

   //Sviluppa la successione di Collatz.
   Collatz(n);

   return 0;
}
```

13.2.6 Palindromo ricorsivo

Il presente esercizio non manipola numeri, ma stringhe, e tende a valutare se una stringa è palindroma o meno. Una parola, una frase o un verso si dicono palindromi se leggendoli da sinistra a destra oppure da destra a sinistra mantengono lo stesso significato. Esempi noti sono i seguenti: osso; kajak; ai lati d'Italia; Angela lava la legna; i topi non avevano nipoti; o mordo tua nuora o aro un autodromo, ecc.

Esercizio - ♢♦♦ Palindromo ricorsivo

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, data una stringa (correttamente dotata di terminatore), valuti ricorsivamente se essa è palindroma o meno.

Soluzione

L'esercizio è un po' diverso dai precedenti e richiede uno sforzo maggiore per ridefinirlo ricorsivamente. Si invita lo studente a non gettare subito la spugna, ma ad insistere tenacemente nella ricerca della soluzione. E' evidente che si dovrà formulare un caso base ed un caso induttivo, ma ad una prima e superficiale analisi non sembra semplice.

Il caso induttivo deve *semplificare* il problema iniziale fino ad ottenere un problema *non ulteriormente semplificabile*, che coincide con il caso base. Quand'è che valutare se una stringa è palindroma o meno è talmente facile che la difficoltà non può essere ulteriormente ridotta? In due casi distinti: 1) se la stringa è formata da una sola lettera oppure 2) se è formata da due lettere uguali. Se quanto appena detto illustra i due casi base, allora non risulta più molto difficile formulare il caso indittivo: una stringa è palindroma se la prima e l'ultima lettera sono uguali e se la stringa fra esse contenuta è palindroma.

Alla luce di quanto detto si può correggere leggermente il concetto di caso base e di caso induttivo e riformulare in maniera ricorsiva la definizione di stringa ricorsiva. Una stringa è palindroma se:

1. è formata da una sola lettera;
2. la prima e l'ultima lettera sono uguali e se la stringa fra esse contenuta è palindroma o nulla.

Il primo caso coincide con il caso base, mentre il secondo con il caso induttivo. Si tratta ora di fare un ulteriore sforzo e rendere il problema un po' più informatico. In fin dei conti si deve arrivare a ipotizzare una funzione avente una lista di argomenti di cui almeno uno deve essere intero e deve "avvicinarsi" alla condizione di terminazione. La stringa potrebbe essere "accorciata" da due puntatori (nel senso di indicatori, non di operatori) che partendo dagli estremi della stringa permettano il confronto e l'avvicinamento. La situazione è illustrata in fig. 13.6.

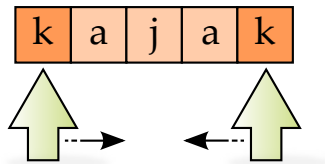



Figura 13.6: Esempio di posizionamento dei puntatori

Indicando con `str[left]` e `str[right]` rispettivamente il carattere puntato dall'indice sinistro della stringa `str` ed il carattere puntato dall'indice destro della stringa è possibile riformulare la definizione ricorsiva in maniera più simbolica:

$$p(str, l, r) = \begin{cases} 1, & \text{se } r - l \leq 1, \\ 0, & \text{se } str[l] \neq str[r], \\ p(str, l+1, r-1), & \text{se } r - l > 1 \wedge str[l] = str[r]. \end{cases} \qquad (13.27)$$

Naturalmente i due puntatori, sinistro e destro, tengono conto del fatto che gli elementi della stringa sono puntati da indici crescenti da sinistra verso destra.

Una possibile funzione, non ancora definitiva, potrebbe essere la seguente:

Listing 13.15: Stringa palindroma 0v70

```
int Palindroma(char *str, int l, int r)
{
   if(l>=r)      //Condizione di terminazione
      return 1; //Caso base
   else
   {
      if(str[l]!=str[r]) //Condizione di terminazione
         return 0;
      else
         return Palindroma(str, l+1, r-1); //Caso induttivo
   }
}
```

Si noti che la condizione `str[l]!=str[r]` può anch'essa essere intesa, a tutti gli effetti, come condizione di terminazione: se detto caso si verifica, infatti, l'algoritmo si interrompe e restituisce un risultato immediato.

Si è accennato al fatto che la funzione illustrata non sia definitiva. In effetti non sono stati ancora affrontati alcuni problemi sostanziali:

1. se la parola fosse stata scritta nella forma *KajAk* anziché kajak, sarebbe stata riconosciuta come palindroma? Una parola scritta in detto modo va considerata palindroma?
2. la frase *Angela lava la legna* verrebbe considerata palindroma dalla funzione? Va considerata palindroma?

Il senso del concetto di "palindromo" invita a considerare palindroma la singola parola indipendentemente dalle maiuscole/minuscole ("il significato deve rimanere lo stesso"), come pure non vanno considerati gli spazi nelle frasi. Alla luce di ciò vanno affrontati detti problemi.

13.2.6.1 *Not case sensitive*

Il confronto fra le due lettere deve essere fatto in maniera tale da ignorare se esse siano state scritte in maiuscolo o minuscolo. Il modo più semplice per implementare detto modo di trattare il confronto è sviluppare una funzione a ciò adibita. Sostanzialmente si possono immaginare due approcci al problema: 1) convertire inizialmente la stringa in lettere tutte maiuscole o tutte minuscole e poi iniziare i confronti, oppure 2) ridurre via via il singolo confronto di due lettere ad un confronto fra maiuscole oppure minuscole.

I due approcci sono sostanzialmente identici: il primo è meno invasivo sul codice fin qui già scritto, ma utilizza maggiori risorse (la stringa va copiata); il secondo non utilizza risorse ma va a modificare leggermente il codice fin qui scritto. Soppesate le due possibilità, si opta per la seconda soluzione.

Come si può confrontare una lettera minuscola con la sua relativa maiuscola? Per rispondere alla domanda può essere utile dare un'occhiata alla tabella visualizzata in appendice A. In essa si nota che le lettere maiuscole si succedono senza eccezioni dalla "A" maiuscola alla "Z" maiuscola e dalla "a" minuscola alla "z" minuscola. Fra i set di caratteri sono frapposti altri caratteri non pertinenti e non appartenenti all'insieme delle lettere dell'alfabeto inglese.

Diventa quindi molto più facile ricondurre una maiuscola ad una minuscola e viceversa. Siccome il problema proposto non è di natura ricorsiva, si propone in fig. 13.7 a fronte il solito diagramma di flusso documentante la soluzione proposta. In tale diagramma si suppone di ricondurre ciascun carattere a minuscolo in modo da rendere omogenei i successivi confronti. Si suppone, momentaneamente, che le stringhe siano effettivamente formate da sole lettere maiuscole e/o minuscole dell'alfabeto inglese.

Nel diagramma si provvede a valutare se il singolo carattere è minuscolo. Ciò avviene per confronto con la lettera 'a' minuscola, che è la prima lettera del secondo set di lettere. Verificando sulla tabella ASCII illustrata in appendice A, ad ogni lettera è abbinato un codice. Nel caso della 'a' minuscola detto codice è 0x61 in notazione esadecimale ovvero 97 decimale. Si assume, quindi, che se il codice abbinato al carattere da valutare è inferiore a 97, il carattere sia forzatamente un carattere maiuscolo. Come ricondurlo a lettera minuscola? E'

sufficiente **sommare** al codice della lettera maiuscola lo spiazzamento fra set di lettere minuscole e set di lettere maiuscole. Si esplica il concetto attraverso un esempio.

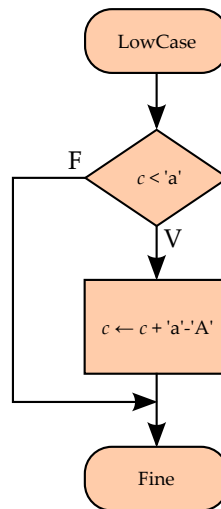



Figura 13.7: *Turn to low case*

Si supponga che la lettera c da valutare sia la 'G' maiuscola. Il fine consiste nel trasformare eventuali lettere maiuscole nelle equivalenti minuscole. Quindi alla fine dell'algoritmo proposto dal diagramma di fig. 13.7, l'algoritmo dovrà ritornare una 'g' minuscola.

Il codice abbinato alla 'G' è il 0x47 esadecimale, ovvero il 71 decimale. Eseguendo il test posto ad inizio dell'algoritmo si evince che la lettera contenuta dal carattere c non è minuscola ($c <' a' = true$), dato che il suo codice è minore di 97. In tal caso, all'attuale valore di c si deve sommare la differenza $'a' -' A'$ ovvero la differenza fra il codice della 'a' (ovvero 97) e quello della 'A' (ovvero 65). Detta differenza vale $97 - 65 = 32$. Sommando 32 al valore di c (ossia 71) si ottiene 103, ovvero 0x67 che, guarda caso, è proprio il codice della lettera 'g' nella tabella ASCII.

Il codice relativo al diagramma di flusso illustrato è il seguente:

Listing 13.16: Funzione LowCase 0v90

```
char LowCase(char c)
{
   if(c<'a')     //Carattere maiuscolo?
      //Il carattere e' maiuscolo
      c += 'a'-'A';

   return c;     //Ritorna il carattere
}
```

Ora è possibile modificare la funzione 13.15 a pagina 345 in modo che essa sia "insensibile" al fatto che la lettera sia maiuscola oppure minuscola. A tal fine è sufficiente modificare la seconda condizione di terminazione sostituendo

le notazioni `str[l]` e `str[r]` rispettivamente con le notazioni `LowCase(str[l])` e `LowCase)str[r])`, come evidenziato nel sottostante codice:

Listing 13.17: Stringa palindroma 0v80

```
int Palindroma(char *str, int l, int r)
{
   if(l>=r)     //Condizione di terminazione
      return 1; //Caso base
   else
   {
      //Condizione di terminazione
      if(LowCase(str[l])!=LowCase(str[r]))
         return 0;
      else
         //Caso induttivo
         return Palindroma(str, l+1, r-1);
   }
}
```

13.2.6.2 Ignorare gli spazi

Rimane da risolvere ancora un'ultima serie di problemi: ignorare gli spazi, le virgole, gli apostrofi, ecc. nelle frasi. Si potrebbe riutilizzare e modificare parzialmente la funzione `LowCase` ponendo, però, attenzione ad alcuni aspetti che potrebbero passare inosservati. Si valuti la seguente frase palindroma:

Avevi visioni d'un evo   ove nudi noi   si viveva

Essa permette quattro tipi di osservazioni:

1. gli spazi non sono palindromi, nel senso che non sono, a destra e a sinistra, alla medesima distanza dal centro della frase;
2. oltre agli spazi si devono ignorare anche altri tipi di caratteri, quali le virgole, gli apostrofi, ecc.;
3. gli spazi potrebbero essere non singoli ma in sequenze di più caratteri;
4. al centro della frase potrebbe esserci una sequenza di spazi formata da più d'un carattere.

Analizziamole uno ad uno.

Che gli spazi non occupino posizioni speculari rispetto al centro è piuttosto ovvio. "Saltare" uno spazio non è obiettivamente cosa di grande difficoltà: è sufficiente spostare di una posizione a sinistra o a destra, a seconda del fatto che il puntatore sia quello di destra o di sinistra.

Intercettare altri tipi di carattere non aggiunge difficoltà al problema: si tratta semplicemente di valutare se il carattere è uno spazio, oppure un apostrofo, oppure una virgola, ecc.

Anche le sequenze di spazi (purché non al centro effettivo della stringa) non rappresentano un problema: è sufficiente spostare il puntatore finché si trova un carattere utile.

L'ultimo punto, pur non rappresentando un'effettiva difficoltà, merita un'analisi supplettiva perché evidenzia che un problema va sempre studiato a fondo, perché potrebbe nascondere insidie che si palesano a funzione già scritta, vanificando magari parte del lavoro fatto. Il "problema" di cui si parla si presenta quando:

1. la stringa, filtrata di ogni carattere spurio che non sia lettera maiuscola o minuscola, è formata da un numero pari di caratteri;
2. quando al centro della stringa si ha una sequenza formata da un numero pari di spazi.

La situazione che si presenta è quella rappresentata in fig. 13.8a.

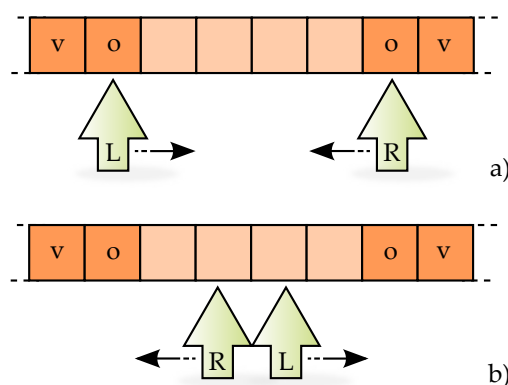



Figura 13.8: Sequenza di spazi a centro stringa

Spostando il puntatore di sinistra verso destra e quello di destra verso sinistra si incontrano due spazi. Ciò costringe entrambi puntatori a spostarsi ancora. In tal modo saranno puntati altri due spazi, per cui i puntatori verranno spostati ulteriormente. Quest'ultima azione farà "incrociare" i puntatori che continueranno a spostarsi alla ricerca di un carattere utile, come evidenziato in fig. 13.8b. A questo punto viene rilevata la condizione di terminazione, anche se in leggero ritardo, e l'algoritmo termina avendo rilevato che la stringa è palindroma. Non si tratta di un vero problema, ma lo si può sostenere solo dopo averlo analizzato.

E' finalmente possibile riscrivere la funzione `LowCase`, magari rinominandola `LowCaseSS` (*Low Case and Skip if Space*) e tenendo conto di quanto detto. Un possibile diagramma di flusso dell'algoritmo è visualizzato in fig. 13.9 nella pagina seguente. Esso si sviluppa in maniera piuttosto semplice, anche se non sono graficamente evidenziati tutti i dettagli (così come dovrebbe effettivamente essere per un diagramma di flusso).

Se il carattere corrente non è una lettera maiuscola o minuscola dell'alfabeto inglese si sposta l'indice (destro o sinistro: la funzione dovrà contenere un parametro che evidenzi di che indice si tratti) finché non si trova effettivamente una lettera.[2]

Se invece il carattere è una lettera, allora si valuta se essa è minuscola o maiuscola. In questo secondo caso si somma lo spiazzamento già illustrato a proposito del diagramma di fig. 13.7 a pagina 347. Si fa effettivamente notare

[2] "Scandalo! Un ciclo, anche se nascosto, in una funzione ricorsiva! L'autore di siffatta blasfemìa va coperto di sputi, legato alla colonna infame ed esposto al pubblico ludibrio!" Ok, si accetta la critica, opponendo una tenue difesa: come al solito non si vuole sommare problema a problema, ma nemmeno togliere tutte le difficoltà allo studente. Per il momento si ammette che la soluzione non è perfettamente elegante e si rimanda ad una soluzione ricorsiva tra breve. Per favore, però: la colonna infame, no.

che la prima parte del diagramma contiene un ciclo. Ciò “inquina” la soluzione ricorsiva togliendole purezza. Si avrà modo di proporre una soluzione

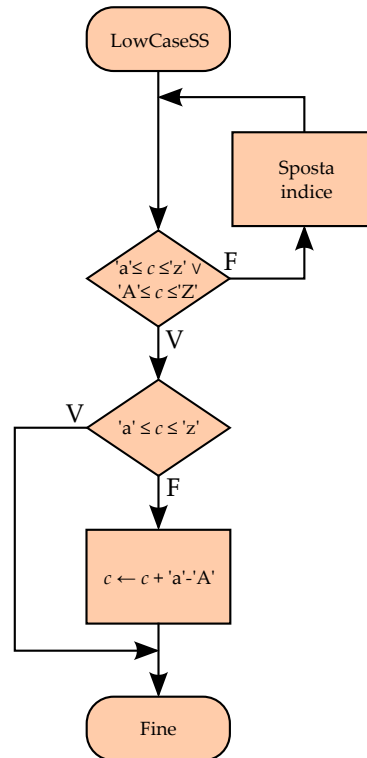



Figura 13.9: *Turn to low case and skip if space*

totalmente ricorsiva in un secondo momento. A tal punto, però, lo studente avrà avuto modo di diventare padrone del problema e sarà in grado di gestire quest’ulteriore complicazione. E’ ora possibile proporre un possibile codice:

Listing 13.18: Funzione LowCaseSS

```
char LowCaseSS(char *c, int *i, int left)
{
   //Valuta se si tratta di una lettera. Altrimenti,
   //se il puntatore e' quello sinistro (left==1),
   //sposta il puntatore a destra.Se il puntatore e'
   //quello destro sposta il puntatore a sinistra.
   //Contestualmente aggiorna anche l'indice *i.
   //Se cio' non venisse fatto Palindroma userebbe
   //degli indici disallineati.
   while(!((*c>='a'&&*c<='z')||(*c>='A'&&*c<='Z')))
      if(left)
      {
         c++;     //Sposta il puntatore
         (*i)++; //Incrementa l'indice sinistro
      }
      else
```

```
            {
               c--;    //Sposta il puntatore
               (*i)--; //Decrementa l'indice destro
            }

         //Valuta se il carattere e' maiuscolo
         if(*c>='A'&&*c<='Z')
            //Il carattere e' maiuscolo
            *c += 'a'-'A';

         return *c;   //Ritorna il carattere
      }
```

La funzione `Palindroma` deve anch'essa essere aggiornata:

Listing 13.19: Stringa palindroma 0v90

```
int Palindroma(char *str, int l, int r)
{
   if(l>=r)     //Condizione di terminazione
      return 1; //Caso base
   else
   {
      //Condizione di terminazione
      if(LowCaseSS(&str[l],&l,1)!= LowCaseSS(&str[r],&r,0))
         return 0;
      else
         //Caso induttivo
         return Palindroma(str, l+1, r-1);
   }
}
```

13.2.6.3 Un ultimo sforzo

Manca ancora la funzione di test e una funzione che ricavi la lunghezza della stringa. Quest'ultima si rende necessaria perché la funzione `Palindroma` necessita, come terzo argomento la lunghezza della stringa.

Vi è quindi la necessità di tracciare un ulteriore diagramma di flusso, non eccessivamente complesso per la verità. Esso è illustrato in fig. 13.10 nella pagina seguente. L'algoritmo utilizza un contatore e suppone di avere in argomento il puntatore ad inizio stringa. l'algoritmo legge ciascun carattere incrementando via via il contatore ad ogni carattere letto finché viene letto il terminatore di stringa (che non viene contato).

La funzione correlata è la seguente:

Listing 13.20: Funzione GetLength

```
int GetLength(char *str)
{
        int l=0;  //Contatore

        while(*str++!=0)
                l++;
        return l;
}
```

La funzione può essere richiamata appena è definita la stringa. A tal proposito si apre una breve parentesi. La funzione standard `scanf` considera lo spazio come terminatore. Ciò significa che se si digita la stringa "I topi non avevano nipoti", la funzione `scanf` memorizza nella stringa solamente l'articolo "I". Il problema è aggirabile utilizzando altre funzioni (ad esempio `gets`, consigliata da MSDN - *Microsoft Developer Network*), che però non si ritiene fondamentale illustrare per i motivi già spiegati nel capitolo 5.

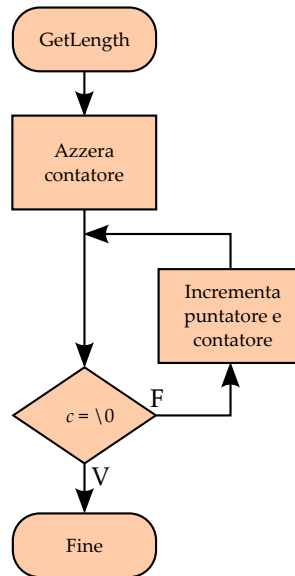



Figura 13.10: Diagramma funzione *GetLength*

E' quindi possibile proporre un possibile programma di test:

Listing 13.21: Test stringa palindroma

```
int main(void)
{
   char s[90]= "Avevi␣visioni␣d'un␣evo␣ove␣nudi␣noi␣si␣viveva";
   int lgt;
   int p;

   //Calcola la lunghezza della stringa
   lgt = GetLength(s);
   p = Palindroma(s, 0, lgt-1);
   if(p==1)
      printf("La␣stringa␣e'␣palindroma");
   else
      printf("La␣stringa␣non␣e'␣palindroma");

   return 0;
}
```

13.2.6.4 Una versione più elegante

Nella sezione 13.2.6.2 si è accennato al fatto che la `LowCaseSS` contenga un ciclo e che detta funzione sia richiamata dalla funzione ricorsiva `Palindroma`. Pur non essendo ciò penalmente perseguibile è però stilisticamente discutibile, dato che ricorsione ed iterazione si contrappongono concettualmente.

Nella presente sezione si pone rimedio alla stortura. Ciò richiede, però, di ripensare la definizione ricorsiva di stringa palindroma proposta nella 13.27 a pagina 345. Alla luce dei ragionamenti progressivi fatti nelle sezioni precedenti, ciò non costa ora molta fatica, ma avrebbe potuto confondere le idee a qualche studente se proposti ad inizio dell'esercizio.

La definizione ricorsiva predetta va dunque integrata, ma non sconvolta. La si potrebbe ridefinire nel seguente modo:

$$p(str,l,r) = \begin{cases} 1, & \text{se } r-l \leq 1, \\ 0, & \text{se } str[l] \neq str[r], \\ p(str,l+1,r), & \text{se } str[l] \neq \text{'a..z'} \wedge str[l] \neq \text{'A..Z'}, \\ p(str,l,r+1), & \text{se } str[r] \neq \text{'a..z'} \wedge str[l] \neq \text{'A..Z'}, \\ p(str,l+1,r-1), & \text{se } r-l > 1 \wedge str[l] = str[r]. \end{cases} \quad (13.28)$$

La differenza è data dai due casi aggiunti. Se il carattere letto non appartiene al set di caratteri da confrontare si richiama ricorsivamente la funzione spostando l'indice, cioè semplificando il problema e contemporaneamente ignorando il carattere. La funzione potrebbe essere la seguente:

Listing 13.22: Stringa palindroma 1v00

```
int Palindroma(char *str, int l, int r)
{
   if(l>=r)     //Condizione di terminazione
      return 1; //Caso base
   else
   {
      //Eliminazione dei caratteri spuri
      if(LowCase(str[l]==' ')
         return Palindroma(str, l+1, r);
      if(LowCase(str[r]==' ')
         return Palindroma(str, l, r+1);

      //Condizione di terminazione
      if(LowCase(str[l])!= LowCase(str[r]))
         return 0;
      else
         //Caso induttivo
         return Palindroma(str, l+1, r-1);
   }
}
```

E' però necessario ridefinire anche la funzione `LowCase`, in modo che possa ritornare il carattere spazio se non viene identificata una lettera. Si ritiene il diagramma di flusso piuttosto banale, per cui si fornisce direttamente la funzione, che è molto simile alla `LowCase` in versione 0v90:

Listing 13.23: Funzione LowCase 1v00

```
char LowCase(char c)
{
   if(c>='A'&&c<='Z')    //Carattere maiuscolo?
      //Il carattere e' maiuscolo
      c += 'a'-'A';

   //Se si tratta di un carattere spurio
   //ritorna uno spazio
   if(!(c>='a'&&c<='z'))
      c = '␣';
   return c;     //Ritorna il carattere
}
```

Con queste ultime aggiunte, il problema può dirsi concluso anche nella sua forma strettamente ricorsiva.

13.2.7 L'algoritmo del contadino russo

Secondo la tradizione popolare russa, esiste un algoritmo, chiamato "algoritmo del contadino russo" che permette la moltiplicazione di due numeri quando almeno uno di essi è un numero naturale.

Esercizio - ◊◊♦ Algoritmo del contadino russo

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dati due numeri $m \in \mathbb{R}$ e $n \in \mathbb{N}$, calcoli il prodotto $m \cdot n$ mediante l'algoritmo del contadino russo, di seguito illustrato:

1. si formino due colonne con m ed n e si scrivano nelle rispettive colonne i valori di m ed n;
2. si scriva una nuova riga, raddoppiando il valore di m e dividendo con divisione intera per 2 il valore di n. Ciò significa che se n è dispari si otterrà un resto di 1 che verrà ignorato;
3. se $n > 0$ si torna al punto 2, altrimenti si sommano tutti i valori di m posti sulla stessa riga dei valori dispari di n. La somma è il prodotto di $m \cdot n$.

Soluzione

Siccome l'algoritmo non è semplicissimo è bene verificarne la validità o, se si preferisce, valutare se lo si è correttamente compreso. Si supponga di

#	m	n
1	1.2	25
2	2.4	12
3	4.8	6
4	9.6	3
5	19.2	1
6	38.4	0

Tabella 13.2: Moltiplicazione mediante l'algoritmo del contadino russo

voler moltiplicare 1.2 per 25, utilizzando l'algoritmo del contadino russo. La tabella 13.2 a fronte illustra lo specchietto di cui parla il testo dell'esercizio.

Le righe 1, 4 e 5 fanno riferimento a valori dispari di n, per cui si devono sommare i rispettivi valori di m, ossia: $1.2 + 9.6 + 19.2 = 30$ che è effettivamente il risultato della moltiplicazione di 1.2 per 25.

Aver verificato la corretta comprensione del problema e del suo algoritmo è fondamentale per proseguire nella ricerca della soluzione. Meno agevole sembra essere la formulazione ricorsiva del problema.

Sembra possibile individuare la condizione di terminazione, che coincide con $n = 0$ e sembra anche possibile associare un calcolo immediato nel caso base: se $n = 0$ il prodotto $m \cdot n$ deve valere 0.

Più difficile sembra essere il caso induttivo. Sicuramente la semplificazione consiste nell'avvicinamento di n alla condizione di terminazione e sembra che ciò avvenga mediante le divisioni per due di n. Concomitante alla divisione per due di n c'è il raddoppio di m. Inoltre, se n è dispari, il valore di m concorre alla somma che forma il risultato finale, ovvero il prodotto $m \cdot n$. Sintetizzando, si potrebbe formulare la seguente definizione ricorsiva dell'algoritmo del contadino russo:

$$f(m,n) = \begin{cases} 0, & \text{se } n = 0, \\ f(2m, n/2), & \text{se } n\%2 = 0, \\ m + f(2m, n/2), & \text{se } n\%2 = 1. \end{cases} \tag{13.29}$$

La definizione fornita è assolutamente corretta. Frequentemente, però, è possibile imbattersi in una definizione un po' più elegante, anche se assolutamente equivalente alla precedente:

$$f(m,n) = \begin{cases} 0, & \text{se } n = 0, \\ f(2m, n/2), & \text{se } n\%2 = 0, \\ m + f(m, n-1), & \text{se } n\%2 = 1. \end{cases} \tag{13.30}$$

Lo studente potrebbe riflettere sulle differenze fra le due definizioni, valutando, ad esempio, quale delle due è più efficiente e veloce. E' possibile scrivere la funzione ricorsiva prendendo a modello, ad esempio, la prima definizione:

Listing 13.24: Algoritmo del contadino russo

```
double Contadino(double m, int n)
{
   if(n==0)  //Condizione di terminazione
      //Caso base
      return 0;
   else
   {
      //Casi induttivi
      if(n%2==0)
         return Contadino(2*m, n/2);
      else
         return m+Contadino(2*m, n/2);
   }
}
```

Una possibile funzione di test potrebbe essere la seguente:

Listing 13.25: Test algoritmo del contadino russo

```
void main(void)
{
   const int kLim = 100; //Limite massimo fattore
   int a, b; //Fattori

   //Definisce i fattori e applica l'algoritmo
   a = DefIntNum(0, kLim);
   b = DefIntNum(0, kLim);
   printf("Il␣prodotto␣di␣%d␣per␣%d␣e'␣%f",a,b,Contadino(a,b));
}
```

13.2.8 Strade di Manhattan

Passeggiare per Manhattan può avere effetti sorprendenti sul lobo frontale dello spensierato passeggiatore, soprattutto se ha un po' di confidenza con gli algoritmi ricorsivi. Ampie zone di Manhattan sono attraversate da due tipi di viali, *Avenue* e *Street*, che hanno una interessante particolarità: le *Avenue* sono tutte parallele fra loro, come pure le *Street*, ma le prime sono perpendicolari alle seconde. Ciò permette una serie di riflessioni ben riassunte dal seguente

Esercizio - ♢♦♦ Strade di Manhattan

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dato un qualunque reticolo $m\ x\ n$, con $m, n \in \mathbb{N} \mid m > 0, n > 0$, rappresentante una porzione della mappa cittadina di Manhattan, calcoli il **numero di percorsi possibili** per andare dal punto A al punto B, rispettando le seguenti regole:

1. sull'asse Est-Ovest è consentita unicamente la marcia in direzione Ovest;
2. sull'asse Nord-Sud è consentita unicamente la marcia in direzione Sud.

Soluzione

Si tratta di un famosissimo problema che permette infinite riflessioni dal punto di vista matematico/informatico e lo studente farà bene a cogliere l'occasione di cimentarsi con il presente esercizio. In fig. 13.11 è illustrato un esempio di reticolo ed un possibile percorcorso rispettoso delle regole fornite.

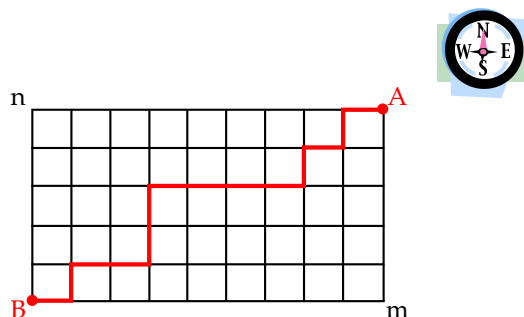



Figura 13.11: Strade di Manhattan

Come al solito, è necessario cercare di definire una condizione di terminazione al problema dato. Ad essa associata deve esserci un caso base che permetta l'immediata formulazione analitica di una soluzione elementare.

L'individuazione del caso base è strettamente collegata all'esistenza di un *cammino minimo unico*. Detto in altri termini: dati due punti A e B su un qualsiasi reticolo $m \ x \ n$, con $m, n \in \mathbb{N} \mid m > 0, n > 0$, esiste un cammino minimo che sia assolutamente unico? Quali sono le condizioni necessarie e sufficienti alla sua esistenza?

Quando il problema formulato si presenta di difficile approccio è sempre conveniente analizzare un sottoinsieme del problema che mantenga intatte le caratteristiche essenziali di quello originale. In tale ottica può risultare utile considerare alcuni casi evidenziati in fig. 13.12.

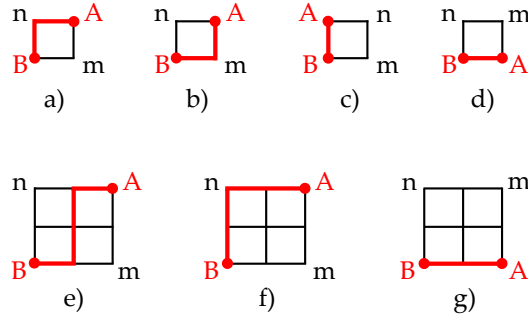



Figura 13.12: Esempi di cammino minimo

Si supponga che il singolo reticolo $1 \ x \ 1$ abbia lato L. Si vuole procedere secondo le citate regole dal punto A al punto B. Esiste un solo cammino minimo? Si valuti a tal fine i due casi evidenziati in fig. 13.12a e 13.12b. In entrambi i casi il cammino è lungo $L + L$ o, se si preferisce, $n + m$. Quindi valutando l'evidenza dei casi a) e b) si può concludere che il percorso minimo dal punto A al punto B *non è unico*. Diversamente, i casi c) e d) illustrano due situazioni in cui il cammino minimo è unico.

Valutiamo la lunghezza del cammino da A a B dell'esempio e). Il primo tratto del cammino è lungo L e si orienta verso Ovest. Poi si percorre un tratto lungo $2L$ verso Sud ed infine un ultimo tratto lungo L verso Ovest. In funzione di n ed m si ha che il cammino c_e vale:

$$c_e = m - (m-1) + n - 0 + (m-1) - 0 = n + m \tag{13.31}$$

dove $m - (m-1)$ corrisponde al primo tratto; $n - 0$ corrisponde al secondo tratto; $(m-1) - 0$ corrisponde all'ultimo tratto. Analoga considerazione si può fare per il cammino illustrato in fig. 13.12f. Il relativo cammino minimo che unisce A a B vale:

$$c_f = m - 0 + n - 0 = n + m \tag{13.32}$$

Anche in questo caso non esiste un cammino minimo unico, ma diversi. Invece, il caso illustrato in fig. 13.12g evidenzia un caso di cammino minimo unico lungo $2L$ ovvero m.

Si può iniziare a formulare l'ipotesi che si abbia più di un cammino minimo, lungo $m+n$, se $m > 0 \wedge n > 0$ e che si abbia un unico cammino minimo, sempre lungo $m+n$, se e solo se $m = 0 \vee n = 0$. Se la suddetta ipotesi fosse vera si potrebbe formulare una possibile condizione di terminazione e relativo caso base.

Ipotizzando il reticolo sovrapposto ad un sistema di assi cartesiani e supponendo il punto B sempre posto nelle coordinate 0,0 (se così non fosse basterebbe traslare i punti A e B fino a far coincidere B con 0,0), lo spazio coperto orizzontalmente dovrà sempre essere m e quello coperto verticalmente dovrà sempre essere n (se si osservano le regole esposte nel testo del problema).

Quindi un qualsiasi cammino coprirà sempre la distanza $m+n$, ovvero

$$c = m - 0 + n - 0 \tag{13.33}$$

Se $m = 0$ il cammino si riduce a

$$c = n - 0 = n \tag{13.34}$$

come pure se $n = 0$ il cammino si riduce a

$$c = m - 0 = m \tag{13.35}$$

Il caso 13.34 documenta uno spostamento puramente verticale, dato che m non può assumere valori negativi. Analogamente il caso 13.35 documenta uno spostamento puramente orizzontale, dato che nemmeno n non può assumere valori negativi. Entrambi i casi assicurano l'unicità del cammino minimo.

Dall'unicità del cammino minimo si passa direttamente all'immediatezza del calcolo. Si può quindi ipotizzare che $m = 0 \vee n = 0$ sia la condizione di terminazione e che il caso base ritorni il valore 1, ovvero l'unicità del percorso.

Con queste premesse anche il caso induttivo non è complicato. Si veda a tal proposito la figura sottostante:

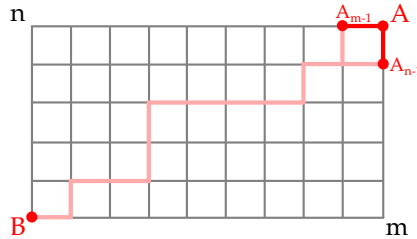



Figura 13.13: Manhattan: caso induttivo

Definire il caso induttivo significa "avvicinarsi" alla condizione di terminazione, ovvero, se si parte dal punto A, sommare il numero di percorsi che si otterrebbero partendo dal punto A_{m-1} e dal punto A_{n-1}. Una possibile definizione ricorsiva diventa quindi la seguente:

$$f(m,n) = \begin{cases} 1, & \text{se } m = 0 \ \vee \ n = 0, \\ f(m, n-1) + f(m-1, n), & \text{se } m > 0 \ \wedge \ n > 0. \end{cases} \tag{13.36}$$

Ora, come al solito, è possibile scrivere il codice della funzione ricorsiva:

Listing 13.26: Manhattan ricorsivo

```
int Manhattan(int m, int n)
{
   if((m==0)||(n==0))  //Condizione di terminazione
      //Caso base
      return 1;
   else
      //Caso induttivo
      return Manhattan(m-1,n)+Manhattan(m,n-1);
}
```

La funzione di test è simile alle precedenti:

Listing 13.27: Test Manhattan

```
int main(void)
{
   const int kLim = 100; //Limite massimo del reticolo
   int m, n; //Limiti del reticolo

   //Definisce i limiti del reticolo
   m = DefIntNum(0, kLim);
   n = DefIntNum(0, kLim);

   //Applica l'algoritmo di Manhattan
   printf("Il numero di percorsi e' %d", Manhattan(m,n);

   return 0;
}
```

13.2.9 Torri di Hanoi

Le torri di Hanoi sono il problema ricorsivo per definizione. Il rompicapo fu inventato dal matematico francese Edouard Lucas nel 1883. Successivamente nacque la leggenda che in qualche monastero dei monaci fossero impegnati a risolvere il rompicapo con 64 dischi e che, a gioco finito, ci sarebbe stata la fine del mondo. La leggenda è totalmente inventata.

Lucas dimostrò che il numero di mosse minimo per completare il gioco è dato dalla relazione $2^n - 1$ dove n è il numero di dischi. Quindi il numero di mosse minimo diventa 18.446.744.073.709.551.615.

Supponendo che i suddetti monaci riescano a fare una mossa al secondo, il gioco finirebbe dopo circa 5.845.580.504 secoli dal suo inizio. E' evidente che in tale ottica diventi piuttosto irrilevante sapere in quale secolo la leggenda ponga l'inizio del gioco.

Esercizio - ♦♦♦ Torri di Hanoi

Si chiede di sviluppare la funzione ricorsiva ed il relativo programma di test che, dati n dischi, con $n \in \mathbb{N} \mid 3 \leq n \leq 10$ (si sconsiglia molto vivamente di utilizzare 64 dischi), documenti mediante stampa a video le mosse da fare per risolvere il gioco delle Torri di Hanoi.

Si ricorda che il gioco utilizza n dischi forati al centro di diametro decrescente più tre pioli e consiste nello spostare i dischi impilati in ordine decrescente

sul piolo A al piolo C (vedi fig. 13.14), mantenendo detto ordine e seguendo, durante gli spostamenti, le seguenti due regole:

1. si può spostare un solo disco alla volta;
2. non si può mai porre un disco di diametro maggiore su un disco di diametro minore.

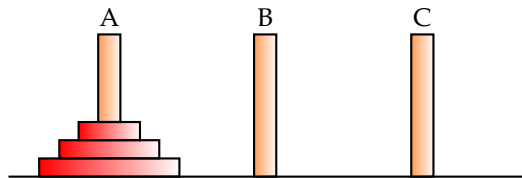



Figura 13.14: Torri di Hanoi

Soluzione

E' sempre buona norma provare un nuovo algoritmo al fine di verificare se lo si è effettivamente capito e interpretato correttamente, ma anche per assimilarlo ed iniziare quel processo, anche inconscio, di analisi che, ineluttabilmente, si dovrà intraprendere tra breve. Una rappresentazione della soluzione per $n = 3$ è data in fig. 13.15.

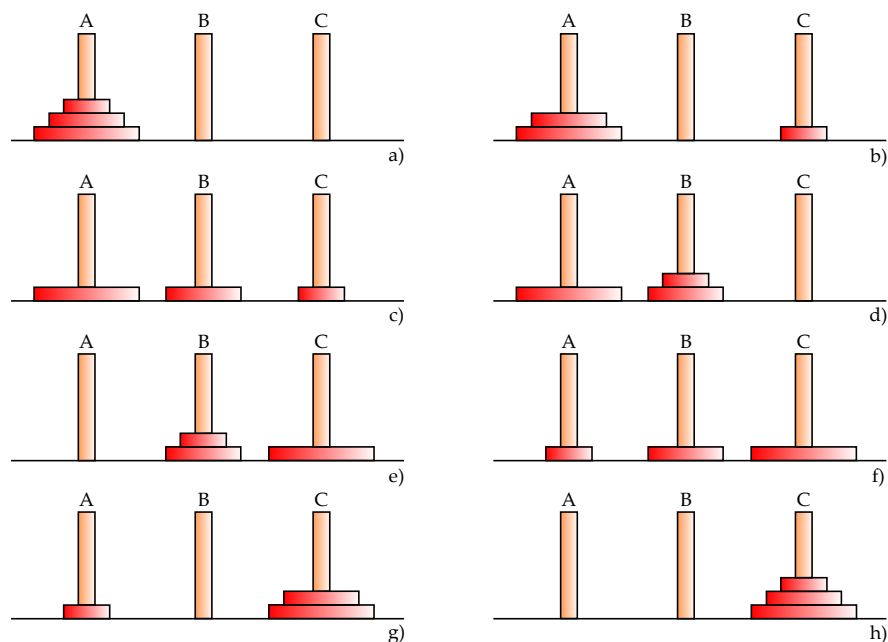



Figura 13.15: Torri di Hanoi: Esempio di soluzione con 3 dischi

La situazione iniziale è rappresentata in fig. 13.15a, mentre le situazioni seguenti, nella sequenza b-c-d-e-f-g-h rappresentano gli spostamenti da eseguire per risolvere il rompicapo secondo le regole date.

Qualche studente, dopo aver visto la soluzione per 3 o 4 dischi cerca di risolvere il problema in maniera iterattiva, incontrando, solitamente, molte difficoltà. Il presente è, invece, il classico problema che, se risolto ricorsivamente, pur mantenendo un'intrinseca difficoltà ineniminabile, risulta di più facile soluzione.

Per fortuna, almeno la condizione di terminazione ed il caso base non costituiscono alcun problema: se il disco è uno solo la soluzione è immediata ed è sufficiente spostare il disco dal piolo di partenza a quello finale.

Anche il caso induttivo non è comunque eccessivamente complicato e può prendere corpo partendo proprio dall'esempio fatto con 3 dischi. La domanda che avremmo dovuto porci sin dall'inizio è la seguente: è risolvibile il gioco con un numero di dischi superiori a 3? Abbiamo visto che con 3 dischi il rompicapo è risolvibile. Induttivamente potremmo ora pensarlo con 4 dischi, ma risolverlo solamente con i primi 3, spostandoli sul piolo B anziché C. Il quarto disco non aggiunge alcuna difficoltà: essendo il primo e più grande disco è come se non ci fosse, dato che gli si può sovrapporre un qualsiasi altro disco. Anche eleggere il piolo B come piolo finale anziché il piolo C non costituisce alcun problema: è sufficiente che il primo spostamento avvenga sul piolo B anziché C.

Una volta che i tre dischi sono stati spostati sul piolo B, è sufficiente spostare il quarto disco sul piolo C e ricominciare da capo spostando i restanti tre dischi dal piolo B a quello C. Così facendo abbiamo dimostrato induttivamente che se si possono spostrare n dischi dal piolo iniziale a quello finale è possibile farlo anche con $n + 1$ dischi e ciò indipendentemente da n.

Così facendo abbiamo descritto proprio il caso induttivo, per cui è possibile definire ricorsivamente l'algoritmo. Prima, però, ancora un'osservazione: fino ad ora abbiamo individuato i tre pioli con le lettere A, B e C. Nella definizione e nella funzione conviene parlare di *piolo di partenza* (p), *piolo finale* (f) e *piolo di mezzo* (m). La definizione ricorsiva per n dischi potrebbe essere la seguente:

$$h(n,p,m,f) = \begin{cases} \textit{sposta disco da } p \rightarrow f, & \text{se } n = 1, \\ h(n-1,p,f,m), h(1,p,m,f), h(n-1,m,p,f), & \text{se } n > 1. \end{cases} \tag{13.37}$$

mediante la quale si può scrivere la funzione ricorsiva:

Listing 13.28: Torri di Hanoi ricorsiva

```
void Hanoi(int n, char from, char free, char to)
{
   if(n==1)  //Condizione di terminazione
   {
      //Caso base
      printf("Sposta␣il␣disco␣dal␣piolo␣");
      printf("%c␣al␣piolo␣%c\n", from, to);
   }
   else
   {
      //Caso induttivo
      Hanoi(n-1,from,to,free);
      Hanoi(1,from,free,to);
      Hanoi(n-1,free, from,to);
}
```

Il programma di test, infine, potrebbe essere il seguente:

Listing 13.29: Test Manhattan

```
int main(void)
{
   const int kLimSup = 10; //Limite massimo dei dischi
   const int kLimInf = 3;  //Limite minimo dei dischi
   int n;                  //Numero di dischi

   //Definisce il numero di dischi
   n = DefIntNum(kLimInf, kLimSup);

   //Applica l'algoritmo delle torri di Hanoi
   Hanoi(n, 'A', 'B', 'C');

   return 0;
}
```

Solitamente lo studente ha bisogno di riflettere sulle successioni dei parametri relativi ai pioli nel caso induttivo della funzione ricorsiva. Per chiarirsi le idee è sufficiente notare che tali successioni ricalcano esattamente quanto esposto prima della definizione ricorsiva del problema, risolvendo il problema per 4 dischi.

13.3 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 13.

Capitolo 14

La programmazione concorrente

Durante lo studio degli algoritmi e dei diagrammi di flusso lo studente ha avuto modo di apprendere che un qualsiasi programma è fondamentalmente formato da tre strutture principali (per dettagli vedi cap. 1, sez. 1.7):

- la sequenza;
- la selezione;
- l'iterazione.

Ciò ha portato l'allievo a descrivere, mediante dette strutture, dei singoli programmi sequenziali durante l'esecuzione degli esercizi proposti. Tali programmi possono, di volta in volta, essere stati anche relativamente complessi, come l'esercizio "Torri di Hanoi" a pag. 360, oppure l'esercizio "Palindromo ricorsivo" a pag. 344. Tutti gli esercizi proposti avevano, però, un comune denominatore: erano rigidamente sequenziali, nel senso che iniziato un compito lo si portava a termine senza iniziarne altri. Questa è una situazione molto comune sui banchi di scuola ma piuttosto rara nei laboratori dove di sviluppano programmi di relativa complessità.

Normalmente un semplice dispositivo è formato da un unico processore *monocore* che coordina ed esegue i diversi compiti a lui attribuiti. Si pensi, ad esempio, ad uno *Switch* Ethernet a 24 canali. Nel più semplice dei casi lo *switch* non dovrà gestire alcun compito (nessun *frame* in transito) se non il semplice controllo di eventuali trame in arrivo, ma nel peggiore dei casi dovrà gestire 24 *frame* che arrivano al dispositivo in maniera totalmente asincrona: per eseguire il *forwarding* lo *switch* deve controllare ciascun flusso di bit a seconda della tecnoclogia scelta (*Store-and-Forward*, *Cut-Through*, *Fragment-Free*, ecc.), verficarne la correttezza e la consistenza al protocollo usato, consultare la tabella CAM e reindirizzare il flusso sulla porta adeguata evitando la collisione delle trame. E si tenga conto che l'esempio è stato fortemente semplificato.

Se, ipoteticamente e per assurdo, lo *switch* avesse un solo canale, lo studente, con molta pazienza e possedendo le opportune conoscenze, potrebbe tracciare un possibile diagramma di flusso dei compiti attribuiti al processore e scoprire che essi sono rigidamente sequenziali. Ma se tutto ciò dovesse avvenire moltiplicato per 24 e in maniera assolutamente asincrona il diagramma si complicherebbe in maniera notevolissima.

Per comprendere il suddetto esempio si immagini di dover descrivere, nei particolari, le azioni che una mamma compie quando deve tener d'occhio i fornelli, mentre il bambino piange e chiede la sua attenzione, e contemporaneamente suona il telefono fisso, il cellulare e il campanello della porta di casa. Tutto questo mentre dal salotto echeggia la voce del marito: "Telefonooo".[1]

Indubbiamente la suddetta mamma avrebbe il suo da fare, ma penso che anche lo studente intento a descrivere dettagliatamente *ciascuna azione che accade "contemporaneamente" ma asincronamente* avrebbe qualche grattacapo.

Il presente capitolo vuole affrontare proprio questo tipo di problemi e la programmazione ad essi correlata. Ciò verrà fatto in modo da far coesistere le seguenti necessità:

- **semplicità**: non è intenzione dell'autore trattare dettagliatamente la programmazione *multithreading* e *multitasking* di un moderno calcolatore, che va oltre gli obiettivi delle presenti pagine, ma introdurre l'argomento evitando le difficoltà maggiori;
- **effettività**: non è nemmeno intenzione dell'autore affogare lo studente in un mare di chiacchere senza fornigli qualche strumento utile ad una programmazione un po' più avanzata;
- **consistenza**: gli argomenti presentati verranno trattati con quel minimo di consistenza teorica che è indispensabile per evitare di fare della "marmellata informatica".[2]

Il corso per il quale i presenti appunti sono stati pensati è un corso di Telecomunicazioni e Informatica con articolazione Telecomunicazioni. Vi è quindi una maggior attenzione alle Telecomunicazioni che all'Informatica e molti aspetti sono indubbiamente visti con una polarizzazione data proprio dall'indirizzo del corso di studi.

Benché mantenere separati i due aspetti impilichi a volte dell'equilibrismo da circo equestre, solitamente la programmazione elaborata da un telecomunicazionista è molto diversa da quella elaborata da un informatico. La prima coinvolge solitamente dispositivi *embedded*, linguaggio C e microcontrollori, la seconda coinvolge di solito PC con sistemi operativi scelti *ad hoc*, linguaggi ad oggetti e microprocessori *multicore*.

Le presenti pagine sono orientate al primo tipo di sistema, dove il microcontrollore è quasi sempre *monocore* e le risorse di memoria volative e non volatile sono minori di *almeno* un fattore 10^6.

Senza dimenticare questa fondamentale premessa, si forniscono alcuni concetti fondamentali.

[1] Non se la prendano i mariti, categoria alla quale, tra l'altro, appartiene anche l'autore: la tendenza allo scherzo, magari anche stereotipato, e alla leggerezza ogni tanto ha il sopravvento.

[2] Dicesi "marmellata informatica" un guazzabuglio di nozioni teoriche che non sfoceranno mai in alcuna applicazione pratica, ma soprattutto obbligano lo studente a studiare a memoria.

14.1 Il processo

Durante un corso di aggiornamento per insegnanti sulle reti di calcolatori l'autore ha avuto modo di sentire dal relatore la seguente definizione:

"Il processo è un omino verde."

La definizione non ha avuto alcun seguito e non è stata arricchita da null'altro, nemmeno da un sorriso ironico. Indubbiamente, ciò che l'insegnante voleva trasmettere era un messaggio provocatorio atto a sottolineare che la definizione di processo non era importante. Fatto sta, che quando all'autore capita di mangiare pesante, la notte sogna orribili omini verdi che dicono minacciosamente di essere dei processi. Per evitare di trasmettere simili incubi agli studenti, si cercherà di entrare maggiormente nel dettaglio dell'argomento, dandogli la dignità che merita.

Naturalmente, lo studente lo avrà capito, si sta parlando di processi nel senso di "compiti" (*task* in inglese), non di "atti giudiziari". Potrebbero, ad esempio, essere quei compiti che stava ipoteticamente eseguendo la mamma nell'introduzione al presente capitolo.

Ad un certo istante i compiti che la mamma era costretta ad eseguire erano "contemporanei", ma siccome la contemporaneità è un concetto, nel migliore dei casi, molto ambiguo, l'informatica ha preferito introdurre il concetto di "parallelismo".[3]

Definizione 16 (Parallelismo)**.**
Capacità di un sistema di iniziare l'esecuzione di un compito prima che compiti precedenti siano stati terminati e di fornire lo stesso risultato che si otterrebbe se i singoli compiti venissero eseguiti sequenzialmente.

Quindi un processo può essere definito nel seguente modo:

Definizione 17 (Processo)**.**
Il processo è un entità astratta definito dalla terna (S, f, s), *dove*

- S *è l'insieme delle variabili di stato (spazio di stato);*
- f *è la funzione di transizione;*
- s *è lo stato iniziale*

e capace di garantire il parallelismo dell'esecuzione.[4]

I concetti di *stato*, *stato iniziale*, *variabile di stato* e *funzione di transizione* dovrebbero essere concetti ormai ben acquisiti dall'allievo durante la frequenza del corso di Sistemi Automatici, ma per quell'unico studente che era a casa col morbillo quando la maestra li ha illustrati, se ne ripropone una rapidissima panoramica.

[3] In Informatica il termine ha assunto negli anni significati sempre più raffinati e che oggi è riservato ai processori *multicore* e più propriamente a quelli con architettura VLIW (*Very Long Instruction Word*). Nelle presenti pagine esso assume un significato più propriamente simile al *multitasking* di cui si vuole evitare momentaneamente di dare una definizione e che sarà oggetto delle prossime sezioni. Detto ciò, sappia lo studente che oggidì i termini *parallelismo* e *multitasking* non possono essere confusi.

[4] Anche in ambito accademico la definizione di processo è sovente accostata a quello di parallelismo, dove quest'ultimo termine è inteso come nella definizione 16. Non si commette, quindi, un errore grave se si tiene a mente quanto detto nella nota precedente. La suddetta definizione è stata elaborata partendo da quella offerta da TISATO-ZICARI in [12]. In tale testo l'argomento è trattato molto dettagliatamente e merita indubbiamente di essere ivi approfondito.

14.1.1 Sistemi e stati

Per poter propriamente parlare di sistemi, stati e funzioni di transizione si deve indubbiamente partire dalla definizione di sistema:

Definizione 18 (Sistema)**.**
Il sistema è un insieme di elementi che interagiscono in modo tale da perseguire un fine comune e impossibile da perseguire per qualsiasi dei suoi elementi individualmente.

Il fatto stesso di aver fornito una definizione suggerisce all'insegnante di dover fornire un esempio esplicativo. Ma per fornirlo il docente dovrà scegliere una fra le diverse forme di rappresentazione e quest'ultima dipende sicuramente dal contesto.

Se si immagina di voler descrivere adeguatamente il sistema "laminatoio" ad una persona completamente a digiuno dell'argomento e per giunta poco abituata ad argomenti di natura tecnica e tecnologica, si opterà probabilmente per una descrizione in linguaggio naturale che illustri il laminatoio come "una macchina industriale atta alla trasformazione di metalli malleabili in lamine o sagomati".

Se la descrizione deve essere sottoposta ad un team di progettisti meccanici al fine di operare delle modifiche meccaniche all'impianto, probabilmente la descrizione più appropriata sarebbe un disegno meccanico quotato della macchina.

Se, invece, si volesse descrivere il processo di laminazione eseguito dal laminatoio, sicuramente sarebbe più appropriato un *diagramma degli stati*.

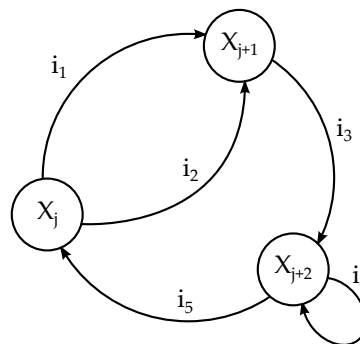



Figura 14.1: Esempio di diagramma degli stati

Mediante tale diagramma è possibile rappresentare

1. l'insieme delle situazioni possibili in cui il processo può trovarsi;
2. la situazione in cui il processo può trovarsi ad un determinato istante;
3. le cause che determinano l'avanzamento del processo;
4. la correlazione fra stato ed uscita.

Se invece dei termini "situazione", "causa" e "avanzamento" utilizziamo i termini "stato", "variabile" e "transazione" il linguaggio utilizzato diventa più appropriato e soprattutto appartenente al modello utilizzato. E' possibile ora cercare di definire le variabili e lo stato:

Definizione 19 (Variabile di stato)**.**
La variabile di stato identifica un dato di input e può assumere uno qualsiasi dei valori appartenenti al suo dominio.

L'insieme di tutte le variabili di stato forma lo *spazio degli stati*.

Definizione 20 (Stato di processo)**.**
Lo stato di processo identifica un particolare momento relativo all'avanzamento del processo.

I singoli stati sono rappresentati nel diagramma di fig. 14.1 mediante dei cerchi. Quindi tale diagramma è formato da tre stati. Gli stati sono individuati univocamente dalle diciture X_{j+n} poste all'interno dei cerchi e può contenere anche la codifica delle uscite. In tal caso nei cerchi compare il simbolismo X_{j+n}/Y_m. A causare le transazioni, ovvero i passaggi da uno stato all'altro, identificati mediante delle frecce orientate, sono le variabili di stato i_q.

Quando una freccia rientra nello stesso stato da cui proviene (si veda la freccia abbinata alla variabile i_4 nella fig. 14.1), si parla di *autoanello* e si dice che lo stato è in *equilibrio* rispetto a tale variabile, dato che un cambiamento del suo valore non produce alcuna transazione ad altro stato.

Non è stato ancora detto (se non in maniera indiretta ed intuitiva) come porre in relazione fra loro variabili, stati, transazioni ed uscite per ottenere il diagramma degli stati. Per fare ciò si dovranno definire due funzioni:

- la funzione delle *transizioni degli stati*;
- la funzione delle *trasformazioni d'uscita*.

La prima è definita simbolicamente come

$$x(t) = f\big(x(t-1), i(t-1)\big) \tag{14.1}$$

che dice che lo stato x nel tempo discreto t (detto anche *stato futuro*) è funzione sia dello stato precedente che del valore che avevano le variabili prima della transizione. Normalmente detta funzione è rappresentata da una *tabella delle trasizioni* e nel diagramma degli stati è rappresentata dalla freccia orientata.

La seconda funzione può assumere due forme leggermente diverse a seconda del fatto che si stia parlando di un *sistema proprio* (macchina di Moore) oppure di un *sistema improprio* (macchina di Mealy). I primi sono caratterizzati dal fatto che le uscite dipendono solamente dallo stato, mentre i secondi dipendono sia dallo stato che degli ingressi. In notazione simbolica si ha:

$$y(t) = g\big(x(t)\big) \qquad \text{(Macchina di Moore)} \tag{14.2}$$

$$y(t) = g\big(x(t), i(t)\big) \qquad \text{(Macchina di Mealy)} \tag{14.3}$$

Siccome è apparso il termine "discreto" è bene chiarire cosa si intenda per *tempo discreto*. Normalmente si contrappone il tempo discreto al tempo continuo, ma sarebbe meglio attingere alla matematica e contrapporre un insieme discreto ad un insieme *denso*.

Definizione 21 (Insieme denso)**.**
Un insieme $A \subset \mathbb{R}$ è detto denso in $\mathbb{R}$ se ogni intervallo aperto non vuoto di $\mathbb{R}$ contiene almeno un elemento d A.

Definizione 22 (Insieme discreto)**.**
Un insieme $B \subset \mathbb{R}$ è detto discreto in $\mathbb{R}$ se non è denso.

Ad esempio un microprocessore non può passare da uno stato all'altro in maniera continua, perché il "tempo" del microprocessore è scandito dal proprio quarzo che permette cambiamenti di stato al micro per periodi di tempo t tali che $t > t_1$, dove t_1, storicamente, è andato via via diminuendo (attualmente è sull'ordine di grandezza dei 10^{-15} secondi), ma sarà sempre una grandezza considerevolmente maggiore di zero.

Ora è possibile definire il concetto di *automa* e *automa a stati finiti*:

Definizione 23 (Automa)**.**
L'automa è un modello di processo a tempo discreto nel quale gli insiemi di input e di output sono finiti.

Definizione 24 (Automa a stati finiti)**.**
L'automa a stati finiti è un automa avente un insieme di stati finito.

In particolare quest'ultima definizione sarà particolarmente utile nelle prossime sezioni.

14.2 Processo e memoria

Dal punto di vista pratico, un processo necessita di memoria per essere gestito. Essa deve poter rendere disponibili in qualsiasi momento le variabili che definiscono lo spazio degli stati, in modo che esse possano essere modificate ed aggiornate prima di ogni transizione. C'è anche la necessità di sapere, in qualsiasi momento, in quale stato si trovi attualmente l'automa a stati finiti abbinato ad un determinato processo, oppure quale sia la sua codifica di uscita, e così via.

Con maggior precisione, si dovrà allocare memoria per contenere i seguenti dati di processo:

- l'identificatore di processo;
- l'*array* delle variabili di ingresso;
- la variabile di stato;
- l'*array* delle variabili di uscita.

Alla luce di quanto detto dovrebbe risultare piuttosto evidente che **ciascun processo necessita di una propria area di memoria**. Tale memoria è solitamente allocata dinamicamente alla creazione del processo e distrutta al termine del processo, ma non si tratta di una regola fissa. Un automa relativamente semplice potrebbe benissimo essere allocato staticamente.

14.3 Il *thread*

Il processo non è un'entità atomica, ma è composta a sua volta da altre entità che concorrono all'esecuzione e avanzamento del processo: i *thread*.

Definizione 25 (Thread)**.**
I thread sono dei flussi sequenziali di istruzioni che possono essere attivati in parallelo all'interno di un processo.

Essi condividono tutti la stessa area di memoria del processo e le stesse strutture dati. Questa definizione può apparire contraddittoria allo studente che probabilmente fa fatica ad immaginare un processore *monocore* che esegue in parallelo due o più "flussi sequenziali di istruzioni". Come ciò possa essere implementato in un microcontrollore verrà illustrato nelle prossime sezioni, per il momento è sufficiente che l'allievo ricordi la definizione 16 di "Parallelismo" data nella sezione 14.1. Una esemplificazione grafica della cooperazione processo-*thread* è data in fig. 14.2:

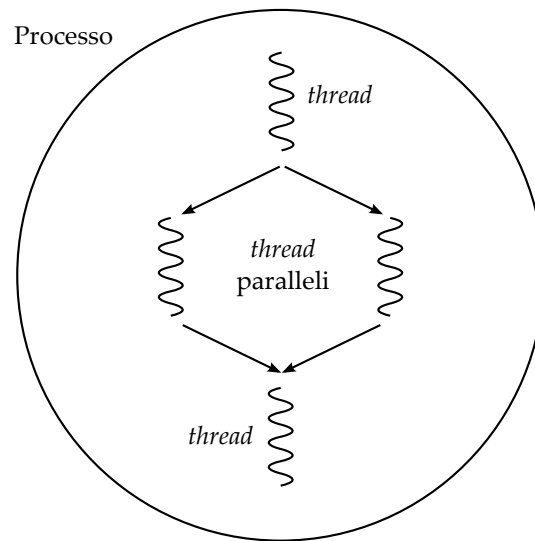



Figura 14.2: Processi e *thread*

14.4 Il problema...

E' venuto il momento di concretizzare quanto visto teoricamente nelle sezioni precedenti. Verrà proposto un problema che implica una programmazione concorrente ed un primo modo molto semplice per risolverlo.

Si suppongano due linee seriali di un sistema *embedded* gestito da microcontrollore. Sulle due porte seriali possono essere presentate due trame che rappresentano altrettanti comandi verso il sistema. Si supponga che:

1. il microcontrollore debba mantenere separate le gestioni dei due processi, ossia la gestione della trama proveniente sulla prima linea seriale da quella proveniente sulla seconda linea seriale;
2. le due trame possano presentarsi sulle due porte in maniera fra loro asincrona, ma che possano coesistere temporalmente (cioè la seconda trama arriva prima che la prima sia stata completamente processata);
3. le due porte si affaccino su due bus, il che implica che ciascuna trama contiene un indirizzo destinazione che va identificato, possibilmente in *real time*;
4. le due trame terminino con altrettanti CRC da processare in *real time*.

Se il processo fosse solo uno il problema sarebbe molto facile da risolvere ed il relativo diagramma di flusso altrettanto elementare:

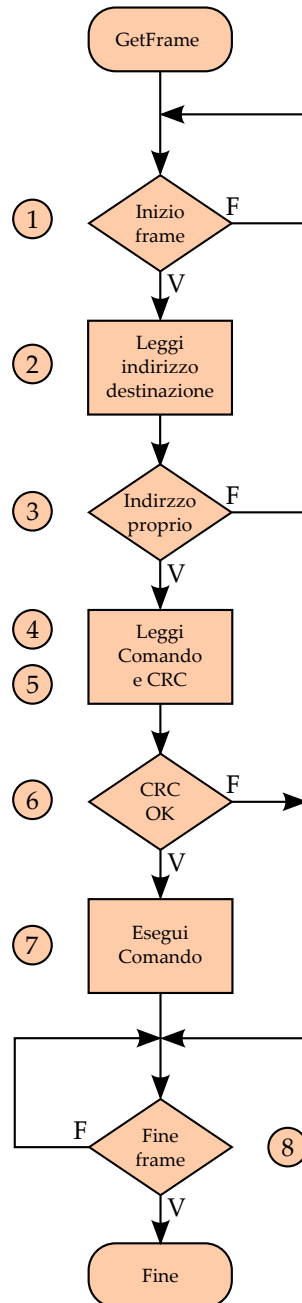



Figura 14.3: Diagramma monoprocesso

E’ molto facile immaginare come il processore possa eseguire sequenzialmente il diagramma di fig. 14.3, ma assai meno banale immaginare che “con-

temporaneamente” sulla seconda porta arrivi una seconda trama che il microcontrollore deve gestire.

La sola idea di dover tracciare un simile diagramma di flusso potrebbe gettare nel panico qualsiasi studente. Si badi che la seconda trama potrebbe arrivare sulla porta seriale quando la prima è già in una qualsiasi delle fasi di gestione del diagramma di flusso di fig. 14.3.

In realtà la soluzione è piuttosto banale anche se richiede un piccolo sforzo di astrazione.

14.5 ...e la soluzione

Il trucco consiste nello sfruttare i “tempi morti”, ovvero quei tempi che il microcontrollore utilizza per attendere un determinato evento.

Supponendo che il protocollo di ricezione sia settato a 57600 baud (molto lento se paragonato ad una rete LAN 100BASETX, ma dignitosa in campo industriale) si ha che i caratteri arrivano circa ogni 17μs. Supponendo che ciascun campo sia composto da almeno 2 byte, il micro deve attendere circa 34μs prima che ci sia un campo da analizzare. Supponendo un quarzo da 40MHz e un ciclo macchina di 100ns, il micro può eseguire circa 340 istruzioni RISC fra l’arrivo di un campo e il successivo. Questi tempi vanno sfruttati.

Si immagini un primo diagramma di flusso come quello di fig. 14.4 dove le parti tratteggiate nelle frecce orientate simboleggiano altri blocchi sequenziali di complessità variabile.

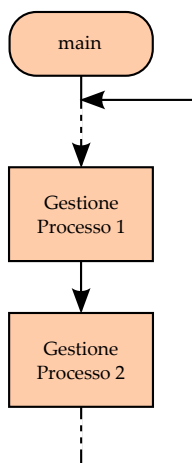



Figura 14.4: Gestione di due processi

Lo studente potrebbe protestare sostenendo che dal grafico appare che i due processi vengono gestiti sequenzialmente. A tale studente bisogna dare atto che ha ragione: d’altronde come potrebbe essere diversamente per un microprocessore *monocore*? Un microcontrollore non può fare altro che eseguire istruzioni sequenzialmente. Però, in certe condizioni, può farlo rispettando la definizione 16 di “Parallelismo” data a pag. 367.

14.5.1 Soluzione mediante automa a stati finiti

Si potrebbe immaginare un automa a stati finiti, identico per i due processi, allocato in due aree di memoria distinte e operante su due strutture dati distinte. L'automa potrebbe essere rappresentato mediante un diagramma degli stati, simile a quello di fig. 14.5

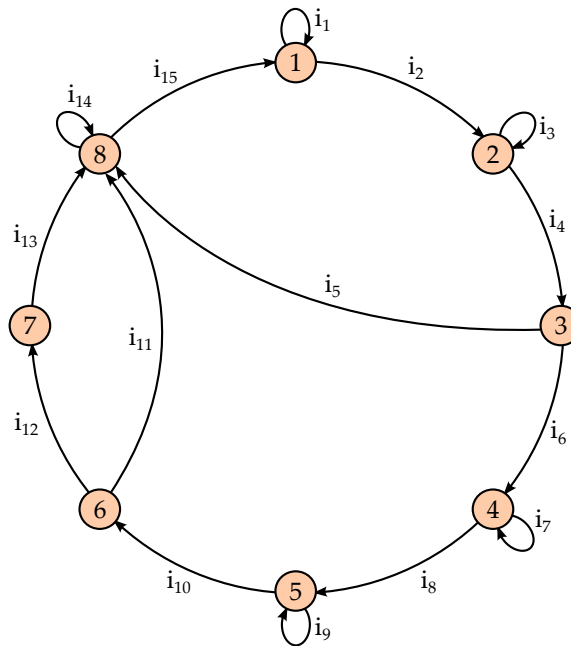



Figura 14.5: Diagramma degli stati automa Processo X

Ipotizzando che lo *stato iniziale* dell'automa sia lo **Stato 1**, il diagramma degli stati si legge nel seguente modo:

1. l'automa rimane in attesa dell'inizio trama;
2. l'automa passa in **Stato 2** ad inizio trama (i_2) e ivi attende (i_3) il completamento dell'indirizzo;
3. ad indirizzo completamente ricevuto ((i_4) si passa in **Stato 3** dove ne viene verificata l'appartenenza. Se l'indirizzo è relativo ad un altro microcontrollore (i_5) si passa in **Stato 8** altrimenti (i_6) si va in **Stato 4**;
4. nel presente stato si attende (i_7) il comando. Quando questo è completamente ricevuto (i_8) si passa in **Stato 5**;
5. qui si attende (i_9) il CRC e appena ricevuto (i_{10}) si va in **Stato 6**;
6. se il CRC è corretto (i_{12}) si va in **Stato 7**, altrimenti (i_{11}) si salta l'esecuzione del comando e si va in **Stato 8**;
7. viene eseguito il comando e al termine (i_{13}) si va in **Stato 8**;
8. si attende la fine della trama (i_{14}) e a trama terminata (i_{15}) si torna nello stato iniziale.

Si confronti ora il diagramma degli stati con il relativo diagramma di flusso evidenziato in fig. 14.6 nella pagina successiva. Il diagramma di flusso evidenzia un aspetto fondamentale che il diagramma degli stati non pone in risalto

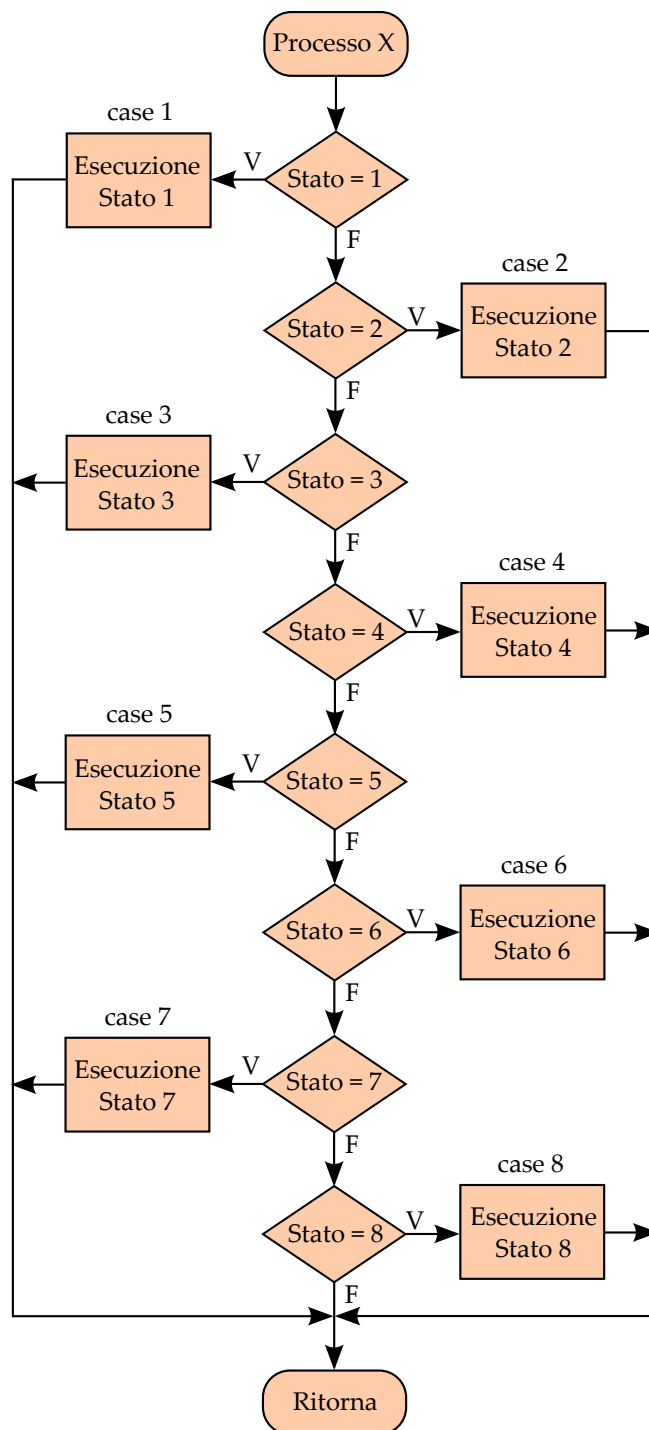



Figura 14.6: Automa a stati finiti del Processo X

(perché non è di sua competenza): quando l'automa è in **Stato 1**, verifica la presenza dell'inizio della trama e se non è ancora arrivata *esce immediatamente* per passare al processo seguente. I cicli di attesa *non esistono*: se c'è qualche azione da eseguire il microcontrollore la esegue, se deve attendere esce. Tale "comportamento" del micro permette al sistema di gestire in parallelo più processi contemporaneamente. Affinché ciò sia possibile, il micro deve conoscere lo spazio degli stati all'atto dell'esecuzione della funzione relativa al diagramma di fig. 14.6 e a tal fine devono essere rispettate le seguenti condizioni:

- deve essere noto lo stato corrente;
- deve essere nota la struttura dati eventualmente associata a tale stato;
- devono essere note le variabili di *input*;
- deve essere nota la tabella delle transizioni;
- deve essere nota la codifica delle uscite.

Si noti che l'esecuzione di ciascuno stato del diagramma di fig. 14.6 corrisponde all'esecuzione di un *thread* e si noti che è possibile anche individuare un esempio di *threading* paralello: in **Stato 5** l'automa rimane in attesa della ricezione del CRC. Il controllo di parità longitudinale ricevuto via linea seriale dovrà essere identico a quello calcolato dal micro localmente, per cui prima di abbandonare lo **Stato 4** il micro può calcolare il CRC locale *mentre* la linea seriale riceve i caratteri del CRC proveniente dall'esterno. Questo è un esempio di parallelismo concreto. Le due azioni (calcolo e ricezione) comportano un effettivo grado di *parallelismo reale*, dato che le due azioni vengono espletate da due parti diverse ed autonome del micro (ALU e SPI), ma anche quando vengono gestite sequenzialmente (durante l'intervento dell'interruzione che annuncia la presenza di un carattere nel buffer di ricezione seriale) appaiono, all'utente esterno, eseguite contemporaneamente.

Al fine di scendere maggiormente nel dettaglio, si propone un esempio di codice in linguaggio C.

14.5.2 Un esempio di automa a stati finiti

Alla base dell'automa a stati finiti c'è una struttura condizionale in forma di `switch`. Essa garantisce un veloce attraversamento della funzione. Si rammenti che si hanno a disposizione al massimo 340 istruzioni assembly (non C!) per ricevere i due caratteri di cui ciascun campo è formato, aggiornare la struttura dati associata allo stato ed eseguire gli eventuali calcoli o comandi.

Un possibile "scheletro"[5] del codice potrebbe essere il seguente:

Listing 14.1: Scheletro Automa a Stati Finiti

```
// Di seguito e' fornito lo scheletro dell'automa a
// stati finiti che implementa il Processo X. L'unico
// parametro della funzione punta alla struttura
// relativa al processo che si suppone essere nota.
// Essa contiene lo stato, l'indirizzo del PC, il crc
// della trama, il comando e l'ID del processo.
int Processo_X(struct Process *p)
```

[5]Si è optato per la traduzione letterale del termine inglese (*skeleton*) usato in Informatica in casi del genere.

```
{
   switch(p->stato)
   {
      case 1: if(FrameStarted(p)) p->stato = 2; break;
      case 2: if(AddressReady(p)) p->stato = 3; break;
      case 3:
      {
         // Verifica l'indirizzo di bus.
         if(GetAddress(p)==p->add)
            p->stato = 4;       // Indirizzo proprio
         else
            p->stato = 8;       // Indirizzo altrui
         break;
      }
      case 4:
      {
         // Attende il comando
         if (CommandReady(p))
         {
            // Comando arrivato: calcola il CRC
            p->stato = 5;
            p->crc = CalcCRC(p);// Multithreading
         }
         break;
      }
      case 5: if(CRCReady(p)) p->stato = 6;     break;
      case 6:
      {
         // Verifica la correttezza del CRC
         if(GetCRC(p)==crc)
            p->stato = 7;
         else
            p->stato = 8;
         break;
      }
      case 7: ExeCommand(p);                    break;
      case 8: if(FrameStopped(p)) p->stato = 1; break;
      default: p->stato = 8;
   }

   // Ritorna lo stato aggiornato;
   return p->stato;
}
```

Tutte le funzioni elencate nel codice non vengono illustrate e si suppongono siano date. A titolo esplicativo esse svolgono le seguenti funzioni:

- `FrameStarted` ritorna 1 se è stato rilevato l'arrivo di una trama;
- `AddressReady` attende l'indirizzo in ricezione sul bus;
- `GetAddress` ritorna l'indirizzo ricevuto dal bus;
- `CommandReady` attende il comando in ricezione sul bus;
- `CalcCRC` calcola il CRC con i dati in possesso;
- `CRCReady` attende il CRC in ricezione sul bus;
- `GetCRC` ritorna il CRC ricevuto dal bus;

- `ExeCommand` esegue il comando;
- `FrameStopped` ritorna 1 se la trama è terminata.

14.5.3 Difetti e pregi dell'automa a stati finiti

L'automa a stati finiti rappresenta una soluzione molto semplice al problema della concorrenza dei processi. In alcuni casi è impossibile prescindere da una qualche forma di programmazione concorrente, più o meno evoluta e l'automa è sicuramente la prima risposta utile ad un problema di tale tipo. Quindi i pregi sono la semplicità e l'immediatezza d'uso. E' sufficiente tracciare un corretto diagramma degli stati del processo (indispensabile qualsiasi sia la forma risolutiva del problema) e scrivere un corretto automa "passante" come quello descritto nella sezione 14.5.2.

Per contro, però, la soluzione mediante automa a stati finiti presenta un evidente difetto: una marcata difficoltà di gestione del *debugging*. Collaudare dei processi asserviti mediante automi passanti è cosa per niente facile e richiede notevole impegno e attenzione. Si tratta di "vedere" un *puzzle* senza poter ricomporre le singole tessere. Si deve, quindi, utilizzare un discreto grado di astrazione si durante la progettazione dell'automa che, soprattutto, durante il suo collaudo.Tale difficoltà può risultare ulteriormente aumentata durante la eventuale condivisione delle risorse, di cui si parlerà nelle prossime sezioni.

Per meglio esplicare il concetto di automa a stati finiti, la prossima sezione è dedicata ad un esercizio *ad hoc*.

14.6 *Led blink* mediante automa a stati finiti

Di seguito viene presentato un esercizio e la relativa soluzione che prevede la gestione concorrente di tre processi piuttosto banali, ma significativi. Il problema verrà risolto mediante l'uso di altrettanti automi a stati finiti, senza implicare l'uso di risorse condivise.

Esercizio - ♢♦♦ *Led blink*

Si supponga un sistema a microcontrollore, basato su PIC16F877, che gestisca tre led (L_1, L_2, L_3) e un pulsante (P_1). Quando si preme il pulsante P_1 il sistema deve far lampeggiare il led L_1 in modo tale da codificare un numero prememorizzato (ad esempio il numero 135) non avente alcuna cifra posta a zero. Il tipo di codifica è indicata in calce. Detto led deve continuare a lampeggiare fino al comando di interruzione. Ripremendo per la seconda volta il pulsante P_1 deve iniziare a lampeggiare il led L_2 e premendo P_1 una terza volta deve lampeggiare il led L_3. Le successive tre volte che si preme il pulsante P_1 si inviano altrettanti comandi di interruzione ai rispettivi led. Quindi premendo una quarta volta P_1 si interrompe il lampeggìo di L_1, premendolo nuovamente si interrompe il lampeggìo di L_2 e così via. Quando tutti i led hanno smesso di lampeggiare, il sistema è pronto per ricominciare da capo la sequenza.

I numeri che i tre led devono visualizzare in codice sono rispettivamente 135, 246 e 1234. Per ovvi motivi di codifica, i numeri non devono contenere la cifra 0. Il sistema potrebbe essere visto come indicatore di un qualche tipo di errore di sistema utilizzato da certi BIOS.

Codice di visualizzazione

Il singolo led visualizza ogni singola cifra del numero lampeggiando tante volte quanto vale la cifra. Ogni singolo lampeggìo deve essere fatto mantenendo il led acceso per 500ms e spento per 500ms. Quando la singola cifra è stata completamente visualizzata mediante detta codifica, il led deve restare spento altri 500ms (serve ad indicare la fine della cifra) e proseguire poi con la successiva cifra. Quando l'ultima cifra è stata completamente visualizzata ed è stata effettuata l'ulteriore attesa di 500ms, si deve aggiungere un ulteriore ritardo di 1000ms (serve ad indicare la fine del numero) e ricominciare da capo. La visualizzazione deve iniziare dalla cifra più significativa.

Soluzione

La complessità del problema risiede nei frequenti e prolungati tempi di attesa. Contemporaneamente tale problema diventa anche una opportunità da cogliere. Il tempo di attesa all'interno di un processo può essere sfruttato per gestire altri processi.

Un altro aspetto, apparentemente innocquo, ma che va affrontato con un po' di metodo, è dato dalla gestione del pulsante P_1.[6] I diagrammi temporali dei processi e dell'attivazione del tasto sono evidenziati in fig. 14.7:

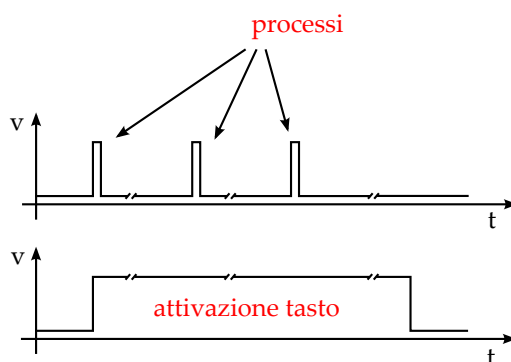



Figura 14.7: Diagramma temporale processi vs. pulsante

L'attivazione manuale di un tasto dura solitamente parecchi millisecondi, mentre la scansione dei tre processi dura poco più di due millisecondi. Se il pulsante non viene gestito in maniera intelligente, data la velocità con la quale i tre processi vengono scanditi, c'è il rischio di attivare e disattivare tutti e tre i processi nell'arco di pochi millisecondi. Ciò va naturalmente evitato.

Quando, durante la gestione del processo 1, si trova il pulsante premuto, si deve eseguire il toggle dello stato del processo e *non non condividere la risorsa relativa al pulsante con altri processi*. In quel momento il processo 1 si è impossessato del pulsante e l'azione eseguita è riferita al solo processo 1. La risorsa (ovvero il pulsante) non deve essere condivisa con altri processi finché essa non ha completamente terminato l'azione. Ciò avviene solo *quando il tasto viene rilasciato*. In quell'istante la risorsa viene liberata e diventa accessibile ad altri

[6] L'utilizzo di un unico pulsante per tutti e tre i led è dovuto al fatto che la *evaluation board* della Microchip che ospita il PIC16F877 non possiede sufficienti tasti da accoppiarli ai rispettivi led.

processi. La prossima volta, quindi che il tasto viene premuto, l'azione sarà abbinata al processo 2, e così via.

L'argomento verrà ripreso in maggior dettaglio quando si parlerà nello specifico della gestione del pulsante.

Se si dovesse scrivere una funzione che esegue la sola codifica del led, senza la gestione del relativo tasto e supponendo che il led sia solamente uno, si potrebbe tracciare il seguente diagramma di flusso:

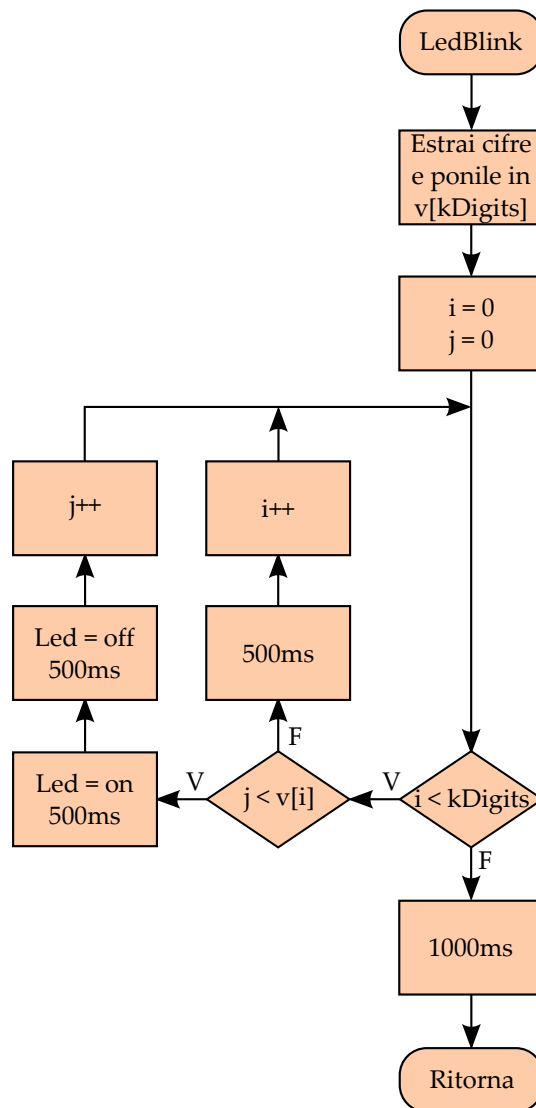



Figura 14.8: Diagramma di flusso LedBlink

Esso documenterebbe un singolo processo piuttosto semplice da implementare. La parte più complessa è probabilmente data dal primo blocco che ha il compito di scomporre il numero abbinato al processo nelle singole cifre e di

porle nel vettore v. Dopodiché, cifra per cifra si fa lampeggiare adeguatamente il led, intervallando ciascuna cifra mediante un ritardo di 500*ms* a led spento. A numero completamente codificato si aggiunge un ulteriore ritardo di 1000ms.

Il codice relativo a detto diagramma di flusso potrebbe essere il seguente:

Listing 14.2: Esempio di *Led blink*

```
void LedBlink(int code)
{
   int j;
   int i;
   int v[kDigits];

   // Estrapola le cifre del codice posto a parametro
   // e le pone nel vettore v in modo che il primo
   // elemento del vettore corrisponda alla cifra piu'
   // significativa del codice.
   for(i=0; i<kDigits; i++)
   {
      // Estrapola la cifra
      d = code%10;
      code /= 10;
      v[kDigits-i-1] = d;
   }

   // Visualizzazione del codice
   for(i=0; i<kDigits; i++)
   {
      // Lo visualizza facendo lampeggiare il led per
      // tante volte quante indica la cifra estrapolata
      // dal codice.
      for(j=0; j<v[i]; j++)
      {
         LedPort |= kLed;
         Delay(k500ms);          // Punto di abbandono 1
         LedPort &= ~kLed;       // Punto di rientro 1
         Delay(k500ms);          // Punto di abbandono 2
      }                          // Punto di rientro 2

      // Mantiene spento il led per altri 500ms
      // (Fine cifra)
      Delay(k500ms);             // Punto di abbandono 3
   }                             // Punto di rientro 3
   // Mantiene spento il led per altri 1000ms
   // (Fine numero)
   Delay(k1000ms);               // Punto di abbandono 4
}                                // Punto di rientro 4
```

In realtà sappiamo che le cose sono molto più complesse, dato che i processi sono tre e devono coesistere. Si tratta, quindi, di scrivere tre automi a stati finiti, possibilmente identici, in modo da doverne scrivere uno solo e poi replicare la soluzione per tre. Una prima cosa da fare è tracciare il diagramma degli stati dell'automa per poi identificare gli stati che prevedono autoanelli. Ad ogni autoanello corrisponde un abbandono dell'automa con successivo prossimo rientro.

Un possibile diagramma degli stati del processo in questione potrebbe essere il seguente:

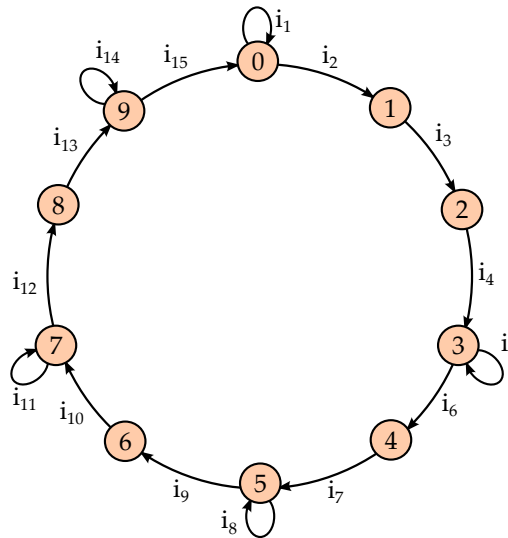



Figura 14.9: Diagramma degli stati automa LedBlink

Si nota che gli autoanelli sono 4, per cui 4 saranno gli stati di attesa. Nell'automa a stati finiti essi rappresentano altrettanti punti di abbandono. Una volta noto il diagramma degli stati, prima del codice vero e proprio, va definita la struttura che contiene i dati del processo:

Listing 14.3: Struttura *Led blink*

```
struct Processo
{
   unsigned char pid;    // ID del processo
   unsigned char on;     // Se = 1, processo attivo
   unsigned int code;    // Numero da codificare
   unsigned char stato; // Stato corrente dell'automa
   int timer;            // Tempo di attesa
   unsigned char digits;// Numero di cifre del numero
   unsigned char v[9];  // Vettore delle singole cifre
   unsigned char mask;  // Maschera di attivazione led
};
```

I campi della struttura sono marcati come `unsigned` (escluso il timer). Ciò è dovuto al fatto che non è necessario trattare numeri con segno, per cui è più prudente dichiarare i campi come sottoinsiemi dei numeri naturali.

Il primo campo è l'identificatore del processo. Nel presente esempio non viene mai usato. Il secondo campo identifica il codice di errore da visualizzare. Il campo `stato` identifica lo stato corrente del processo, mentre gli ultimi due campi rappresentano rispettivamente il numero di cifre del numero da visualizzare e il vettore delle singole cifre.

Definita la struttura, ora è possibile scrivere il relativo codice del singolo automa:

Listing 14.4: *Led blink* mediante automa a stati finiti

```
void ASFLedBlink(Processo *p)
{
   int d,i=0;

   // Gestione dello stato
   switch(p->stato)
   {
      case 0:   // Se il numero vale zero, esce.
         {
            if(p->code>0)
               p->stato = 1;    // Numero > 0
            break;
         }
      case 1:   // Estrae le cifre del numero
      {
         // Estrae le cifre del codice posto a parametro
         // e le pone nel vettore v in modo che il primo
         // elemento del vettore corrisponda alla cifra
         // piu' significativa del codice.
         p->digits = 0;
         do
         {
            // Estrae la cifra
            d = p->code%10;
            p->code /= 10;
            p->v[p->digits] = d;
            p->digits++;
         }
         while(p->code>0);
         p->stato = 2;          // Nuovo stato
      }
      case 2:   // Accende il led e lancia un ritardo di
                // 500ms
      {
         LedPort |= p->mask;    // Accende il led
         p->timer = 500;        // Lancia il timer
         p->stato = 3;          // Nuovo stato
         break;
      }
      case 3:   // Attende la fine dei 500ms
      {
         if(p->timer<=0)
            p->stato = 4;       // Nuovo stato
         break;
      }
      case 4:   // Spegne il led e lancia un ritardo di
                // 500ms
      {
         LedPort &= ~(p->mask); // Spegne il led
         p->timer = 500;        // Lancia il timer
         p->stato = 5;          // Nuovo stato
         break;
      }
```

```
            case 5:   // Attende la fine dei 500ms
            {
               if(p->timer<=0)
               {
                  // Valuta se la cifra e' terminata (e se il
                  // il numero e' terminato)
                  if(--(p->v[p->digits]))
                     p->stato = 2;    // No: continua
                  else
                  {
                     p->digits--;
                     p->stato = 6;    // Fine cifra
                  }
               }
               break;
            }
            case 6:   // Lancia un ulteriore ritardo di 500ms
                      // (fine cifra)
            {
               p->timer = 500;        // Lancia il timer
               p->stato = 7;          // Nuovo stato
               break;
            }
            case 7:   // Attende la fine dei 500ms
            {
               if(p->timer<=0)
               {
                  // Fine numero?
                  if(p->digits)
                     p->stato = 2;    // No: continua
                  else
                     p->stato = 8;    // Fine numero
               }
               break;
            }
            case 8:   // Lancia un ulteriore ritardo di 1000ms
                      // (fine numero)
            {
               p->timer = 1000;       // Lancia il timer
               p->stato = 9;          // Nuovo stato
               break;
            }
            case 9:   // Attende la fine dei 1000ms
            {
               if(p->timer<=0)
                  p->stato = 1;       // Ricomincia da capo
               break;
            }
         }
      }
```

Si nota con facilità che l'automa è molto più complicato (soprattutto da capire) della relativa funzione che simula il processo. E quel che è peggio, l'automa è molto difficile da collaudare: è molto facile che eventuali errori passino

inosservati. La difficoltà di collaudo nasce dal fatto che le combinazioni con le quali i singoli processi asincroni possono correlarsi fra loro è piuttosto alta e non sempre vengono studiate e collaudate tutte.

Ora è possibile occuparsi della gestione del pulsante P_1. Anche in questo caso è possibile rappresentare la gestione del tasto mediante un diagramma degli stati:

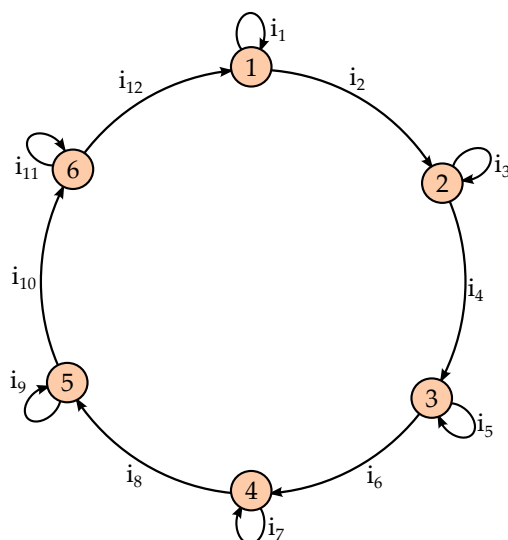



Figura 14.10: Diagramma degli stati automa ManagePB

Esso si può leggere nel seguente modo:

1. in stato 1, l'automa rimane in attesa che il pulsante venga premuto;
2. quando si verifica l'evento i_2 (tasto premuto), si passa in stato 2. In tale stato si esegue il *toggle*[7] del processo 1. Finché il tasto è premuto (i_3), si permane in stato 2;
3. quando il punsalte viene rilasciato (i_4) si passa in stato 3 rilasciando la risorsa occupata dal processo 1. Da questo momento in poi la risorsa (il pulsante) può essere utilizzato da altri processi. Iin stato 3 si permane finché il tasto non viene nuovamente premuto;
4. quando il tasto è premuto (i_6), si passa in stato 4. La risorsa, ora è occupata dal processo 2, di cui si esegue il *toggle*. In stato 4 si permane finché il tasto rimane attivo (i_7);
5. quando il tasto viene finalmente rilasciato si passa in stato 5. La risorsa viene anch'essa rilasciata dal processo 2 e diventa nuovamente libera;
6. quando si preme per la terza volta il pulsante i_{10} si passa in stato 6 e si esegue il *toggle* del processo 3. In tale stato si permane finché il tasto è premuto. Quando il tasto è rilasciato si ritorna in stato 1 e si ricomincia da capo.

[7]Ossia attiva il processo se è disattivo, oppure disattiva il processo se è attivo.

Dal diagramma degli stati si può passare direttamente al codice senza tracciare il diagramma di flusso.[8]

Listing 14.5: Gestione del tasto

```
int ManagePB(Processo *p1,
             Processo *p2,
             Processo *p3,
             int statoPulsante)
{
   // Trattandosi di un automa, il codice inizia
   // con uno switch.
   switch(statoPulsante)
   {
      case 1:
      {
         // Se il tasto e' premuto esegue il toggle
         // del processo 1 e passa in stato 2.
         if(PBPort&kPBMask)     // Testa bit 4 del PORTA
         {
            p1->on = ~p1->on;
            statoPulsante = 2;
         }
         break;
      }
      case 2:
      {
         // Passa in stato 3 quando il pulsante
         // viene rilasciato.
         if(!(PBPort&kPBMask))  // Testa bit 4 del PORTA
            statoPulsante = 3;
         break;
      }
      case 3:
      {
         // Se il tasto e' premuto esegue il toggle
         // del processo 2 e passa in stato 4.
         if(PBPort&kPBMask)     // Testa bit 4 del PORTA
         {
            p2->on = ~p2->on;
            statoPulsante = 4;
         }
         break;
      }
      case 4:
      {
         // Passa in stato 5 quando il pulsante
         // viene rilasciato.
         if(!(PBPort&kPBMask))  // Testa bit 4 del PORTA
            statoPulsante = 5;
         break;
      }
      case 5:
```

[8] Non si tratta, dal punto di vista della documentazione, di un errore: il diagramma degli stati è effettivamente sufficiente per documentare un automa.

```
            {
               // Se il tasto e' premuto esegue il toggle
               // del processo 3 e passa in stato 6.
               if(PBPort&kPBMask)     // Testa bit 4 del PORTA
               {
                  p3->on = ~p3->on;
                  statoPulsante = 6;
               }
               break;
            }
            case 6:
            {
               // Passa in stato 1 quando il pulsante
               // viene rilasciato.
               if(!(PBPort&kPBMask))  // Testa bit 4 del PORTA
                  statoPulsante = 1;
               break;
            }
         }
      }
```

Ora è possibile scrivere il main avente il compito di richiamare i singoli processi. Esso potrebbe essere simile al seguente:

Listing 14.6: main con tre processi

```
#define PBTris  TRISA
#define LedTris TRISB
#define PBPort  PORTA
#define LedPort PORTB
#define kPBmask 0x10

int main(void)
{
   Processo p1, p2, p3; // Dichiarazione processi
   int statoPB = 1;     // Stato del pulsante

   // Inizializzazione degli I/O
   PBTris = 0xFF;
   LedTris &= 0xF0;

   // Inizializza le maschere dei led abbinate ai processi
   p1.mask = 0x01;      // Processo 1 = RB0
   p2.mask = 0x02;      // Processo 2 = RB1
   p3.mask = 0x04;      // Processo 3 = RB2

   // Inizializzazione dei processi
   p1.code = 135;       // Primo numero
   p2.code = 246;       // Secondo numero
   p3.code = 1234;      // Terzo numero
   p1.pid = 1;          // Identificatore p1
   p2.pid = 2;          // Identificatore p2
   p3.pid = 3;          // Identificatore p3
   p1.stato = p2.stato = p3.stato = 0;
   p1.on = p2.on = p3.on = 0;
```

```
    for(;;)
    {
       // Ritardo di circa 1ms. Il sottostante codice
       // implica la conoscenza della frequenza di
       // clock e delle caratteristiche del micro. E'
       // assolutamente da sconsigliare. La nostra
       // attenzione va, pero', ai processi e non
       // al ritardo (che andrebbe gestito mediante
       // interruzione);
       for(int i; i<1000000; i++);

       // Aggiorna i timer
       p1.timer--;
       p2.timer--;
       p3.timer--;

       // Gestione pulsante
       statoPB = ManagePB(&p1, &p2, &p3, statoPB);

       // Gestione processi
       if(p1.on) ASFLedBlink(&p1); // Processo 1
       if(p2.on) ASFLedBlink(&p2); // Processo 2
       if(p3.on) ASFLedBlink(&p3); // Processo 3
    }

    return 0;            // Esce
}
```

14.6.1 Il collaudo di *Led blink*

Come accennato precedentemente la difficoltà reale non sta tanto nella scrittura del codice, pur complesso per certi versi, quanto nel suo collaudo. Effettuare un collaudo esaustivo di processi multitasking è cosa tutt'altro che semplice, perché sposta il problema dal processo al sistema operativo, nel presente caso dal processo all'automa.

Un primo passo da effettuare è il collaudo del singolo processo, ma ciò non è sicuramente sufficiente: una volta wassicurato il buon funzionamento dell'automa relativo al singolo processo, si deve effettuare il collaudo, ben più oneroso, della coesistenza di più processi e quindi anche di più automi. La difficoltà consiste nella totale asincronicità dei processi, per cui, ad esempio, il processo 2 potrebbe essere attivato quando il processo 1 sta elaborando il secondo stato, oppure quando il processo 3 sta elaborando il sesto stato.

Dal punto di vista teorico tutti i casi possibili andrebbero testati. Se il processo ha 9 stati, la coesistenza di due processi identici prevede il collaudo di 9 situazioni differenti: 1) il processo 2 inizia durante l'esecuzione dello stato1 del processo 1; 2) il processo 2 inizia durante l'esecuzione dello stato 2 del processo 1; ecc. Se i processi identici sono 3 si dovrebbero collaudare 81 situazioni diverse, e così via. Se i processi non sono perfettamente identici (nel nostro caso i tre processi sono differenti, perché i tre numeri hanno cifre e lunghezze diverse), le cose si complicano ulteriormente.

Il collaudo deve quindi avvenire ad un livello di astrazione superiore rispetto al solito, dato che quasi sempre non è possibile eseguire un collaudo esaustivo. Ciò richiede un notevole grado di attenzione e di impegno da parte del collaudatore.

Al variare della complessità dell'automa a stati finiti e, soprattutto, del numero di processi, aumenta quindi la difficoltà di *debugging*. Esiste, però, una seconda soluzione al problema della programmazione concorrente che trasferisce la difficoltà del problema al "sistema operativo", rappresentato, in questo caso, dalla soluzione stessa che verrà illustrata nella prossima sezione. Il *debugging* viene fortemente semplificato a fronte di un notevole aumento della difficoltà della soluzione generale del problema.

14.7 Parallelismo mediante controllo di flusso

Una seconda forma di programmazione concorrente è realizzabile mediante *controllo di flusso*. Si tratta di un tipo di soluzione più complicato rispetto all'automa a stati finiti, però anche molto più elegante e facile da collaudare. E' adatto a sistemi complessi ove coesistono numerosi processi tutti asincroni fra loro. Inoltre, si tratta di un tipo di soluzione che si adatta bene ai tre vicoli di semplicità, effettività e consistenza proclamati nell'introduzione del presente capitolo.

E' bene, però che lo studente percepisca sin d'ora la difficoltà dell'argomento, in modo da poter organizzare quelle difese indispensabili per fronteggiare un argomento ostico.[9]

La programmazione concorrente su un sistema monoprocessore e *monocore* si realizza concretizzando un'idea piuttosto semplice: sfruttare i "tempi morti" di un processo per far avanzare altri processi. Qualsiasi sia il metodo utilizzato per implementare la soluzione alla concorrenza utilizza, in un modo o nell'altro, la suddetta idea.

Il concetto di "controllo di flusso" è in sé piuttosto semplice, più difficile è la sua implementazione: quando il flusso delle istruzioni incontra dei cicli di attesa, *si esce dal processo per rientrare, in un secondo momento, nell'esatto punto in cui lo si era abbandonato.* Detto così sembra semplice, ma è in realtà piuttosto complesso. ll motivo di tanta complessità risiede nel fatto che devono essere demoliti i pilastri della programmazione strutturata con buona pace del teorema di separazione di Böhm e Jacopini (vedi sezz. 8.1.1 e 8.1.2).

Il ciclo di attesa, infatti, può trovarsi in un qualsiasi punto del codice e mentre "abbandonarlo" è di relativa facilità, "rientrarvi" nelle stesse condizioni dell'abbandono implica la "ricostruzione" dell'intero ambiente di rientro. Si cercherà di dissipare la nebulosità di quest'ultimo concetto con un esempio.

14.7.1 Abbandono e rientro

Si supponga l'esercizio appena visto nella sezione precedente. Tre led devono lampeggiare, all'attivazione di un tasto, visualizzando tre distinti codici di errore. I tre led devono lampeggiare "contemporaneamente". Si supponga anche

[9] Solitamente quelle difese sono indicate in ambito scolastico con il termine "studio".

che, momentaneamente, il processo sia uno solo, ben asservito dalla funzione `LedBlink` illustrata nel codice 14.2 a pagina 381 che, per comodità si riproduce di seguito:

Listing 14.7: Funzione *Led blink*

```
void LedBlink(int code)
{
   int j;
   int i;
   int v[kDigits];

   // Estrapola le cifre del codice posto a parametro
   // e le pone nel vettore v in modo che il primo
   // elemento del vettore corrisponda alla cifra piu'
   // significativa del codice.
   for(i=0; i<kDigits; i++)
   {
      // Estrapola la cifra
      d = code%10;
      code /= 10;
      v[kDigits-i-1] = d;
   }

   // Visualizzazione del codice
   for(i=0; i<kDigits; i++)
   {
      // Lo visualizza facendo lampeggiare il led per
      // tante volte quante indica la cifra estrapolata
      // dal codice.
      for(j=0; j<v[i]; j++)
      {
         LedPort |= kLed;
         Delay(k500ms);         // Punto di abbandono 1
         LedPort &= ~kLed;      // Punto di rientro 1
         Delay(k500ms);         // Punto di abbandono 2
      }                         // Punto di rientro 2

      // Mantiene spento il led per altri 500ms
      // (Fine cifra)
      Delay(k500ms);            // Punto di abbandono 3
   }                            // Punto di rientro 3
   // Mantiene spento il led per altri 1000ms
   // (Fine numero)
   Delay(k1000ms);              // Punto di abbandono 4
}                               // Punto di rientro 4
```

Per semplicità i cicli di attesa atti a ottenere i ritardi di 500*ms* sono stati scritti sotto forma di funzione (`Delay`) il cui unico parametro indica il numero di millisecondi di ritardo che il codice implementa. Sono anche indicati i punti in cui il codice andrebbe abbandonato ed i rispettivi punti in cui si dovrebbe rientrare nel codice se si volesse implementare una programmazione concorrente.

Si supponga ora di eseguire il codice della suddetta funzione fino al "Punto di abbandono 1". Arrivati a quel punto si dovrebbe attendere per 500*ms*, dopo

aver acceso il led, in modo tale che la sua accensione sia ben visibile. E' evidente che il micro non deve "fare nulla": non deve eseguire calcoli particolari, né gestire periferiche o altre risorse relativamente al processo in corso, ossia la gestione del led. Quindi, sarebbe utile valutare se altri processi necessitano della gestione del microprocessore e abbandonarlo momentaneamente, per 500*ms*.

Durante quel periodo il micro può effettivamente gestire, ad esempio, un secondo o terzo led, ciascuno con un proprio codice. Potrebbe ad esempio estrarre le singole cifre e accendere il led per il primo lampeggìo. Passati, però, i 500*ms* il micro deve rientrare nel primo processo nel "Punto di rientro 1". Ma rientrare in tale punto (già di per sé difficile) non serve a nulla se non si sono memorizzati i valori delle variabili `j`, `i` e `v` e se non si sono ricaricati detti valori nelle stesse variabili.

L'allievo tenga conto che il metodo che verrà scelto per fare tutto ciò deve essere il più semplice e trasparente possibile. Non è pensabile, ad esempio, attuare mille strategie diverse, una per ciascuna funzione e una per ciascun numero di variabili posssibile.

Insomma, si tratta di un bel problema. Non si tratta, quindi, semplicemente di violare di teorema di separazione, ma di

1. **riconoscere** una situazione (numero, tipo e successione delle variabili locali; numero tipo e successione dei parametri della funzione; tipo dell'eventuale valore di ritorno);
2. **congelare** la situazione memorizzando in memoria tutte le variabili e tutti i parametri;
3. **abbandonare** la funzione ricordandosi del punto di abbandono, senza alterare lo *stack*;
4. **rientrare** nella funzione quando è scaduto un tempo oppure è disponibile una risorsa, sempre senza alterare lo *stack*;
5. **scongelare** la situazione e ricaricare le variabili e i parametri nel giusto ordine;
6. **riprendere** l'esecuzione delle istruzioni fino al prossimo punto di abbandono oppure fino alla fine della funzione.

Alcune delle azioni elencate sembrano di una relativa semplicità: la memorizzazione dei valori delle variabili e dei parametri, ad esempio. Molto più complesso è il riconoscimento del numero delle variabili e dei parametri e dei relativi tipi. Infine abbandonare una funzione trovandosi, ad esempio, nel bel mezzo di un ciclo `for` nidificato e posto dentro un ciclo `while` per poi rientrare esattamente nello stesso identico punto e, per giunta, senza alterare lo *stack*, sembra, francamente, piuttosto difficile.

Le prossime sezioni cercheranno di guidare lo studente attraverso questo arduo compito. Si cercherà di essere il più chiari possibile, ma lo studente deve sapere che dovrà spendere molta fatica per padroneggiare i concetti che gli verranno presentati.

Prima di entrare nel vivo dell'esposizione è bene prendere subito dei riferimenti. Tutto il codice che seguirà nelle prossime pagine è riferito in generale alla famiglia dei dsPIC33F della Microchip ed in particolare al micro dsPIC33FJ32MC202.

Il codice è stato testato anche sui microcontrollori dsPIC33FJ64MC804 e dsPIC33FJ256MC710A, ma non sugli altri micro della famiglia dsPIC33F, anche

se non c'è motivo di dubitare del corretto funzionamento del software. Inoltre, il codice presentato non ha la pretesa di essere un software professionale di programmazione concorrente, ma semplicemente un esempio didattico di programmazione un po' più avanzata. Molti aspetti di una effettiva programmazione concorrente (ad esempio la ricorsione concorrente) non verranno esaminati, perché al di là degli obiettivi dei presenti appunti e perché richiederebbero un numero di ore non proponibile per essere trattati esaustivamente.

14.7.2 La libreria setjmp.h

Il linguaggio C non prevede alcun tipo di interfaccia utente né di *utility* matematiche, di trattazione stringhe, gestione della memoria, ecc. Per tale motivo esso è sempre stato affiancato, dall'ANSI C in poi, da librerie che facilitassero il lavoro del programmatore in vari campi. Una di tali librerie standard è la `setjmp` (la relativa intestazione si chiama `setjmp.h`), che risolve il problema dell'abbandono e del rientro. La libreria fornisce due funzioni utili all'esecuzione di "salti non locali" ovvero trasferimenti del controllo di flusso da una funzione all'altra senza l'alterazione dello *stack*. Tale pacchetto software ha il compito di agevolare il programmatore nella trattazione delle eccezioni, ma ha trovato utile impiego anche nella realizzazione di codice concorrente.

14.7.2.1 La funzione setjmp()

La `setjmp()` è la prima delle due funzioni implementanti i salti non locali. Il suo prototipo è il seguente:

Listing 14.8: Funzione setjmp()

```
int setjmp(jmp_buf env);
```

La funzione setta il buffer `env` (*environment*) con i dati utili al microprocessore per il successivo rientro. Solitamente il produttore del software non rilascia documentazione particolareggiata, ma si può inferire dal contesto d'uso che vengono salvati, almeno, i valori relativi:

- al Program Counter;
- allo Status;
- allo Stack Pointer,

ma con ogni probabilità vengono salvati anche altri registri interni del micro. Comunque il produttore dichiara che i dati salvati sono sufficienti per un corretto ripristino del contesto abbandonato.

Prima di utilizzare la funzione `setjmp()` è necessario aver dichiarato precedentemente il buffer `jmp_buf` in modo che lo spazio in memoria sia già correttamente allocato. Detto buffer occupa uno spazio in memoria esiguo (72 byte per il dsPIC33FJ64MC202).

14.7.2.2 La funzione longjmp()

Una volta salvato lo stato del micro (*Program Counter*, *Status*, registri fondamentali, *Stack Pointer*, ecc.) e abbandonata la funzione che contiene il tempo di attesa, si deve rientrare nello stesso identico punto della funzione nelle stesse identiche condizioni di PC, *Stack*, ecc.

Ciò avviene mediante la funzione `longjmp`, che permette di saltare all'indirizzo memorizzato, recuperando lo stato del microcontrollore. Si noti che il salto della `longjmp` può essere fatto anche attraverso livelli di *stack* differenti e attraverso diversi livelli di strutture di programmazione (cicli, istruzioni condizionali, ecc.). La funzione `longjmp)` è dichiarata come seguen:

Listing 14.9: Funzione longjmp()

```
void longjmp(jmp_buf env, int val);
```

Questa capacità di saltare attraverso diversi livelli di strutture e *stack* viene sfruttata negli automi a stati finiti. Mediante le suddette funzioni, infatti, si può abbandonare in qualsiasi punto l'automa e rientrare esattamente nell'istruzione successiva in un secondo tempo.

Affinché il meccanismo sia completamente trasparente è, però, necessario salvare anche le variabili locali della funzione/automa come pure i suoi eventuali argomenti. Ciò avviene memorizzando i dati relativi alle variabili ed agli argomenti nello *heap* prima di effettuare il salto mediante la `setjmp`.

14.7.2.3 Il buffer di memorizzazione

Salvare il contesto di un micro prima del salto significa salvare la minima quantità di registri necessaria a descriverlo compiutamente. Si è accennato al fatto che detto contesto è salvato in un buffer di 72 byte. Tale buffer è dichiarato di tipo `jmp_buf`, come nel codice sottostante:

Listing 14.10: Il buffer di memorizzazione

```
typedef int jmp_buf[_NSETJMP];
```

La costante è dichiarata nel file `yvals.h` incluso a `setjmp.h` e alla fine di quella che sembra essere una vera e propria caccia al tesoro si evince che `_NSETJMP` vale 16+2, per cui vengono allocati $(16+2) \cdot 4 = 72$ *byte* nel buffer di memorizzazione `jmp_buf`.

14.7.2.4 L'uso di setjmp() e longjmp()

Senza un esempio concreto non è semplice intuire l'uso che si può fare delle due funzioni `setjmp()` e `longjmp()`. Prima di illustrare un esempio classico si deve sapere che richiamando la `setjmp()` si memorizza il contesto del micro nel buffer `jmp_buf` dopodiché la funzione ritorna 0.

Richiamare, invece, la `longjmp()` setta i registri del micro con i valori salvati nel buffer `jmp_buf`. Ciò significa che, dopo l'esecuzione della `longjmp()`, *non viene eseguita l'istruzione immediatamente seguente la* `longjmp()` *ma quella che segue la* `setjmp()` e che il valore ritornato dalla `setjmp()` non sarà più 0, ma `val`. Un esempio banale ma concreto di quanto detto è il seguente:

Listing 14.11: Esempio banale di uso di setjmp() e longjmp()

```
#include <setjmp.h>

void main(void)
{
   jmp_buf env; // Buffer di memorizzazione
   int ris;
```

```
    ris = setjmp(env);           // Punto 1
    printf("ris␣=␣%d\n", ris);   // Punto 2

    // Esce dal main se ris non e' 0
    if (ris != 0)
       exit(0);

    longjmp(env, 1);             // Punto 3
    printf("Questa␣frase␣non␣verra'␣mai␣stampata\n");
}
```

La prima istruzione ad esssere eseguita dopo la sezione dichiarativa è quella indicata al punto 1, ovvero la `ris = setjmp(env);`. Tale istruzione salva il contesto del micro nel buffer indicato da `env`, dopodiché si ritorna dalla funzione con 0 in `ris`. L'istruzione seguente, indicata al punto 2, stampa il contenuto di detta variabile, per cui si ottiene a video la scritta `ris = 0`. Siccome il valore ritornato vale 0 *non* si esce dal main e si prosegue nell'esecuzione dell'istruzione `longjmp(env, 1);` indicata al punto 3.

Tale istruzione *ricarica nel micro il contesto* salvato nel buffer `env`. Si noti che ciò comporta che *non verrà mai eseguito alcun* `return` *dalla* `longjmp()`, *perché il PC e lo stack pointer vengono alterati e ripristinati al valore che avevano all'atto della memorizzazione dei registri nel buffer* `env`.

Quindi l'istruzione posta sotto la `longjmp` non verrà mai eseguita, come recita la stessa stringa di stampa, bensì si ritornerà al punto 1 e si uscirà dalla `setjmp()` con il valore indicato in secondo argomento dalla `longjmp()`, ossia con 1. Ciò viene documentato al punto 2 dalla stampa, che riporterà a video la stringa `ris = 1`.

Stavolta l'istruzione condizionale, riconoscendo una condizione booleana vera, permette l'uscita dal main. Una introduzione maggiormente dettagliata dell'argomento è data in PLANK [21].

La particolare caratteristica delle due funzioni `setjmp()` e `longjmp()` permette una gestione intelligente degli automi a stati finiti, anche se dette funzioni non sono stete espressamente pensate per tale uso. Lo studente non è, però, ancora pronto per affrontare tale studio, ma deve prima conoscere un'altra libreria piuttosto particolare: la `stdarg.h`.

14.7.3 La libreria stdarg.h

La libreria standard `stdarg.h` permette alla funzioni di accettare un numero indefinito di argomenti e di tipo non noto. Funzioni di tal tipo sono dette funzioni *variadic*. Originariamente pensata per rendere flessibili le macro, si adatta perfettamente agli scopi prefissi dalla presente sezione.

La sintassi che permette alle funzioni *variadic* di accettare un numero variabile di argomenti è regolata dai *tre puntini*, come si seguito evidenziato:

Listing 14.12: Funzione *variadic* corretta

```
int Pippo(int a, char b, ...);
```

Tali funzioni devono, però, avere almeno un parametro, per cui la sottostante dichiarazione non è corretta:

Listing 14.13: Funzione *variadic* errata

```
int Pluto(...); // Argomenti mancanti
```

14.8 Esercizi

Gli esercizi riportati nelle seguenti pagine sono tutti relativi a quanto esposto nel capitolo 14.

1. ◊◊◊ Si fornisca una possibile definizione di *parallelismo*.
2. ◊◊◊ Si fornisca una possibile definizione di *processo*.
3. ◊◊◊ Si fornisca una possibile definizione di *sistema*.
4. ◊◊◊ Si fornisca una possibile definizione di *variabile di stato*.
5. ◊◊◊ Si fornisca una possibile definizione di *stato di processo*.
6. ◊◊◊ Si fornisca una possibile definizione di *stato futuro*.
7. ◊◊◊ Si fornisca una possibile definizione di *macchina di Moore*.
8. ◊◊◊ Si fornisca una possibile definizione di *macchina di Mealy*.
9. ◊◊◊ Si fornisca una possibile definizione di *tempo discreto*.
10. ◊◊◊ Si fornisca una possibile definizione di *automa*.
11. ◊◊◊ Si fornisca una possibile definizione di *automa a stati finiti*.
12. ◊◊◊ Si fornisca una possibile definizione di *thread*.
13. ◊◊◊ Un'area di memoria abbinata a un processo deve contenere quali variabili minime?
14. ◊◊◊ Qual è l'idea centrale che la programmazione concorrente sfrutta per poter funzionare?
15. ◊◊◊ Qual è la struttura del linguaggio C solitamente usata per implementare un automa a stati finiti?
16. ◊◊◊ Si tracci il diagramma degli stati relativo al diagramma di flusso di fig. 14.3 a pagina 372.
17. ◊◊♦ Si tracci il diagramma degli stati relativo alla ricezione di un singolo *frame* in protocollo HDLC.
18. ◊◊♦ Si tracci il diagramma degli stati relativo alla ricezione di una serie di *frame* in protocollo *Stop-and-Wait*.
19. ◊◊♦ Si risolva autonomamente l'esercizio "LedBlink" proposto nella sezione 14.6. Più precisamente:
    (a) si tracci il diagramma degli stati del singolo processo;
    (b) si tracci il diagramma di flusso del singolo processo;
    (c) si delinei la struttura in memoria abbinata al singolo processo;
    (d) si codifichi in linguaggio C l'automa a stati finiti di tre processi concorrenti;
    (e) si testi il codice usando l'ambiente di sviluppo MPLabX.

20. ◊◊♦ Si operi la seguente variante al predetto esercizio:

    - tre led devono lampeggiare rispettivamente alla frequenza di 2Hz, 1Hz e 0.5Hz;
    - il primo led deve iniziare a lampeggiare quando il pulsante P1 viene attivato una prima volta. il secondo led deve iniziare a lampeggiare quando P1 viene attivato una seconda volta e il terzo led deve iniziare a lampeggiare quando P1 viene attivato per la terza volta.

    Al fine di impostare la risoluzione nel modo corretto si eseguano i seguenti passi nell'ordine proposto:

    (a) si tracci il diagramma degli stati del singolo processo;
    (b) si tracci il diagramma di flusso del singolo processo;
    (c) si delinei la struttura in memoria abbinata al singolo processo;
    (d) si codifichi in linguaggio C l'automa a stati finiti di tre processi concorrenti;
    (e) si testi il codice usando l'ambiente di sviluppo MPLabX.

21. ◊◊♦ Si consideri la seguente variante all'esercizio precedente.

    - se si attiva brevemente il pulsante P1 una prima volta durante il lampeggìo dei tre led, il primo led viene spento e messo in pausa. Ripremendo P1 il primo led riprende a lampeggiare. Lo studente è libero di far lampeggiare il led sincronicamente o meno rispetto al lampeggìo precedente alla pausa;
    - ripremendo P1 brevemente, si mette in pausa e si riattiva il secondo led e poi il terzo led;
    - premendo P1 per più di un secondo i tre led si spengono e deve essere possibile ricominciare il processo da capo.[10]

22. ◊♦♦ Si ricerchino in rete le frequenze delle note della scala cromatica e si piloti il buzzer posto sulla linea RC2 della scheda PICDEMO in modo da "suonare" la melodia di "Fra Martino". Essendo un canone, ogni volta che si preme il tasto (a tempo, dopo ogni battuta, naturalmente), deve essere introdotta una ulteriore voce.

Figura 14.11: Fra Martino

[10]Lo studente non si concentri unicamente sul risultato (ossia il funzionamento del codice) ma sull'impostazione dell'intero problema, seguendo la sequenza di passi proposta dall'insegnante. Può risultare utile confrontare fra studenti i diagrammi prodotti, ad esempio postandoli sul Pronto Soccorso del sito Moodle.

Appendice A

Il codice ASCII

ASCII è un acronimo che significa *American Standard Code for Information Interchange*. La sottostante tabella ASCII contiene i caratteri non visibili (dal primo al 31*esimo*) che hanno funzioni di controllo del testo (o della trama) ed i cosidetti caratteri tipografici dal 32*esimo* carattere in poi.

La colonna più a sinistra della tabella indica i 4 bit meno significativi del carattere, mentre la prima riga indica i 3 bit più significativi del carattere.

<table>
<tr><th>LSB/MSB</th><th>000</th><th>001</th><th>010</th><th>011</th><th>100</th><th>101</th><th>110</th><th>111</th></tr>
<tr><td>0000</td><td>NUL</td><td>DLE</td><td>SP</td><td>0</td><td>@</td><td>P</td><td>′</td><td>p</td></tr>
<tr><td>0001</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td></tr>
<tr><td>0010</td><td>STX</td><td>DC2</td><td>″</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td></tr>
<tr><td>0011</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td></tr>
<tr><td>0100</td><td>EOT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td></tr>
<tr><td>0101</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td></tr>
<tr><td>0110</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td></tr>
<tr><td>0111</td><td>BEL</td><td>ETB</td><td>′</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td></tr>
<tr><td>1000</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td></tr>
<tr><td>1001</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td></tr>
<tr><td>1010</td><td>LT</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td></tr>
<tr><td>1011</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td></tr>
<tr><td>1100</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td></tr>
<tr><td>1101</td><td>CR</td><td>GS</td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td></tr>
<tr><td>1110</td><td>SO</td><td>RS</td><td>.</td><td>></td><td>N</td><td>∧</td><td>n</td><td>∼</td></tr>
<tr><td>1111</td><td>SI</td><td>US</td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td></tr>
</table>

Tabella ASCII (7 bit code)

Appendice B

Little Endian e Big Endian

Un argomento che può presentare talvolta aspetti di scarsa chiarezza o di ambiguità è legato al modo in cui i microprocessori ed i sistemi in generale gestiscono la scrittura in memoria dei singoli byte, con particolare riferimento alla memorizzazione dei numeri interi.

Si supponga un microprocessore o un microcontrollore avente un parallelismo superiore agli 8 bit, ad esempio, 32 bit. Esso ha due modi distinti per scrivere in memoria una word: in modalità *little endian* oppure *big endian*, a seconda della filosofia adottata dal costruttore. Si veda, a tal proposito la figura sottostante:

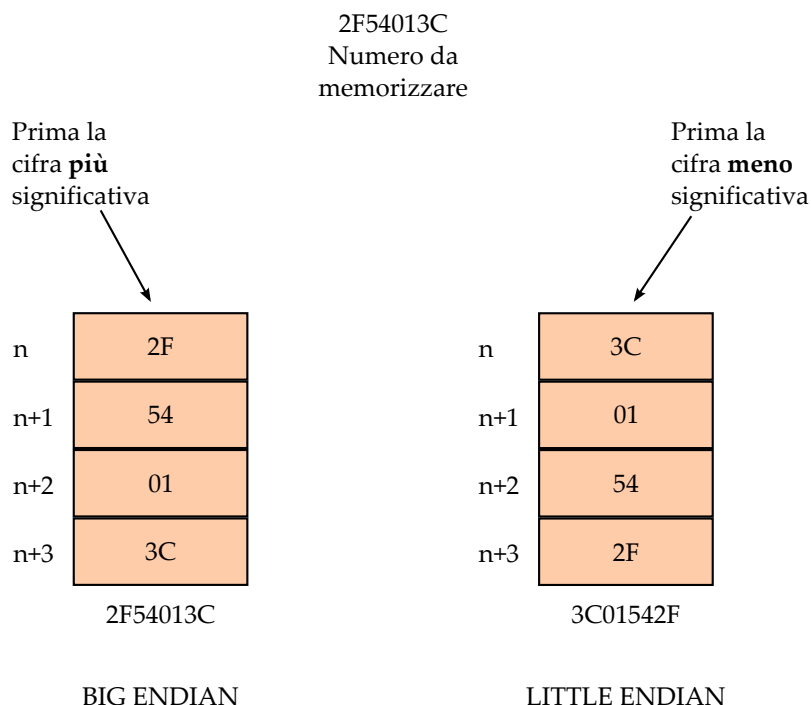



Figura 14.12: Endiannes

Si noti che esiste un problema di *endianness* anche utilizzando processori a 8 bit. Si supponga, infatti, di compilare un programma, ad esempio, in linguaggio C: i numeri interi verranno compilati utilizzando una delle due endianness, anche in questo caso secondo la scelta operata dal costruttore del compilatore.

Il modo dell'informatica, quindi, non è per nulla d'accordo sull'uso di questi due standard e ha dato vita a due veri e propri fronti contrapposti[11] in proposito, come evidenziato nella tabella sottostante. I fogli tecnici dei processori

Big Endian	Little Endian
Motorola	Intel
Apple	IBM
Sun	
Bus VME	Bus PCI
Protocolli Internet	

Tabella 14.1: Big Endian Vs. Little Endian

come quelli dei compilatori forniscono comunque informazioni esaustive sul tipo di endianness scelto dal costruttore, aiutando, in tal modo, il progettista ad interpretare correttamente il dato numerico.

[11] I termini *little endian* e *big endian* derivano dal romanzo di Jonathan Swift "I viaggi di Gulliver". Nel romanzo due popolazioni che vivono sulle isole di Lilliput e Bluefuscu si fanno la guerra perché hanno adottato due modi diversi di rompere le uova: dalla estremità più grande (ossia big endian – Lilliput) oppure dall'estremità più piccola (ossia little endian – Bluefuscu). D'altronde anche l'autore concorda sul fatto che "*tutti i veri credenti rompano le uova dall'estremità conveniente*".

Bibliografia

[1] Boole George, *Indagine sulle leggi del pensiero*, Einaudi, 1976

[2] Deitel Harvey, Deitel Paul, *C Corso completo di programmazione*, Ed. italiana per Apogeo, 2000

[3] Fagarazzi Bruno, Mialich Roberto, Rossi Giancarlo, *Sistemi Automatici*, Voll. 1-3, Calderini, 1994

[4] Joint Technical Committee ISO/IEC JTC 1, *ISO/IEC 9899 Programming languages - C*, Draft Edition, Settembre 2007

[5] Hancock Les, Krieger Morris, *Il linguaggio C*, McGraw Hill, Settembre 1988

[6] Kanigel Robert, *L'uomo che vide l'infinito*, Rizzoli, Febbraio 2003

[7] Kernighan Brian Wilson, Ritchie Dennis MacAlistair, *The C programming language*, Prentice Hall, Second Edition 1985

[8] Knuth Donald Ervin, *The Art of Computer Programming*, Voll. 1-3, Addison Wesley, Third Edition 1997

[9] Odifreddi Piergiorgio, *Classical Recursion Theory*, Vol. 1 Paperback, 1992

[10] Sedgewick Robert, *Algoritmi in C*, Addison-Wesley, 1993

[11] Swift Jonathan, *I viaggi di Gulliver*, Mondadori, 2003

[12] Tisato Francesco, Zicari Roberto, *Sistemi Operativi: architettura e progetto*, Clup, 1985

[13] Wirth Niklaus, *Algoritmi + Strutture Dati = Programmi*, Tecniche Nuove, 1989

Articoli

[14] Goldberg David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, March 1991

[15] Sacks Gerald Enoch, *Teoria della ricorsività*, Seminario Matematico Vol. 55, 1 - Università Politecnico Torino, 1997

[16] Testa Frank, *AN575 - IEEE 754 Compliant Floating Point Routines*, Microchip Technology Inc., 1997

Sitografia

[17] Joint Technical Committee ISO/IEC JTC 1, *ISO/IEC 9899 Programming languages - C* Committee Draft consultabile al sito http://www.open-std.org/JTC1/SC22/WG14/

[18] Fernando Artigiano, *Lorem Ipsum* consultabile all'indirizzo http://www.it.lipsum.com

[19] Microchip Technology Inc., *Sito Microchip* consultabile all'indirizzo http://www.microchip.com

[20] Odifreddi Piergiorgio, *Corso di logica matematica* consultabile al sito http://www.piergiorgioodifreddi.it/video/corso-di-logica-matematica

[21] Plank Jim, *CS360 Lecture notes - setjmp* consultabile al sito http://web.eecs.utk.edu/~huangj/cs360/360/notes/Setjmp/lecture.html

[22] WebCite, *Real Programmers don't use PASCAL* consultabile all'indirizzo http://www.webcitation.org/659yh1oSh

Indice analitico

Indice degli esercizi